Event Driven HA Simulator Reference Manual

Department of Computer Science

Stony Brook University

Mike True

September 23rd, 2006

Table of Contents

1. Background Information

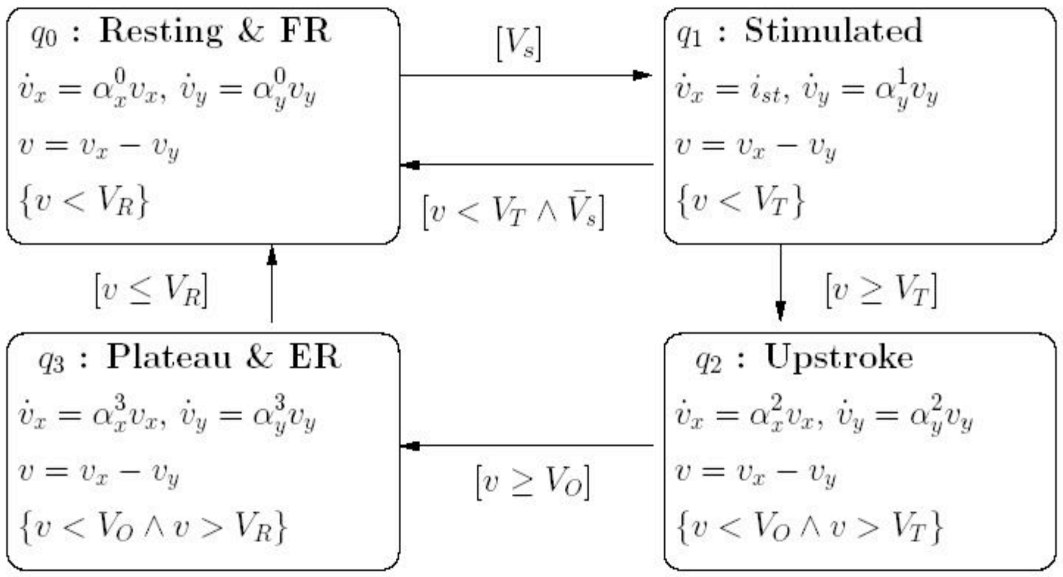
1.1. Biological Background and Hybrid Automata

Excitable cells are a type of cells that respond to electrical stimuli with electrical signals known as Action Potentials (AP). Each AP is fired by a cell as an all-or-nothing response to stimuli external to the cell. The behavior of an AP is generally independent of the magnitude of the stimuli which caused it. A couple of examples of excitable cells found in mammals include those found in cardiac tissue and neurons.

Over the course of the past century, various mathematical models have been introduced as representations for various types of excitable cells. These models often make use of complex differential equations, rendering efficient simulations based directly off of the models very difficult. In [1], we proposed a Hybrid Automata (HA) representation for these mathematical models which separated the phases of each AP into states and associated linear differential equations with each state for the behavior of the variables while the cell is in that state. We experienced a significant improvement in simulation time due to the relative simplicity of the linear differential equations with respect to the complex equations found in the original models.

Each HA consists of a series of states where each state contains a set of associated differential equations that control how the variables change with time while the HA is in that particular state. These states can have transitions to other states that occur whenever the transition's guard conditions are met. Each state may also have invariant conditions, although in our HA models we ensure that each state's invariants are always met by assigning the appropriate guards to each of the state's transitions. Fig. 1.1.1 shows an example of a Hybrid Automata, which in particular is a representation of the Hodgkin-Huxley model.

The HA shown in Fig. 1.1.1 only takes into consideration a single-cell model and does not consider the effect of neighboring cells such as the ones found in tissues of cells. Section 1.2 discusses the principles behind a simple HA-based implementation of a simulator for a tissue of excitable cells. By observing that the excitable cells are non-responsive to external stimulation in the *upstroke* and *plateau & ER* states, we can develop an event-driven model, discussed in section 1.3, which is capable of obtaining a speedup factor of more than 5 with respect to the simpler HA-based implementation. The remainder of this paper will cover information and usage instructions for our C implementations of these simulators.



q_0 : Resting & FR
$\dot{v}_x = \alpha_x^0 v_x,\ \dot{v}_y = \alpha_y^0 v_y$
$v = v_x - v_y$
$\{v < V_R\}$
$[V_s]$
q_1 : Stimulated
$\dot{v}_x = i_{st},\ \dot{v}_y = \alpha_y^1 v_y$
$v = v_x - v_y$
$\{v < V_T\}$
$[v < V_T \wedge \bar{V}_s]$
$[v \leq V_R]$
$[v \geq V_T]$
q_3 : Plateau & ER
$\dot{v}_x = \alpha_x^3 v_x,\ \dot{v}_y = \alpha_y^3 v_y$
$v = v_x - v_y$
$\{v < V_O \wedge v > V_R\}$
$[v \geq V_O]$
q_2 : Upstroke
$\dot{v}_x = \alpha_x^2 v_x,\ \dot{v}_y = \alpha_y^2 v_y$
$v = v_x - v_y$
$\{v < V_O \wedge v > V_T\}$

Fig 1.1.1. The HA representation for the Hodgkin-Huxley model

1.2. A Time-step Integration Implementation

The process of transforming our single-cell HA model specifications into code is fairly trivial. We only need to update our variables using time-step integration in accordance with the current state's associated differential equations and check the guards to determine when a transition is required. The implementation of our diffusion models is more complex because we must to take into account the effect of neighboring cells on a particular cell's voltage and we need to determine when these changes coupled with externally applied stimuli constitute a stimulation event. As previously mentioned, we make the assumption that these factors play a negligible role during the *upstroke* and *plateau & ER* states and we will only concern ourselves with them during the *resting & FR* and *stimulated* states.

To obtain the updated values for the variables v_x, v_y, and v_z associated with a cell in the *resting & FR* and *stimulated* states, we first calculate the sum of all currents involved and use it to determine the cell's target voltage. The three sources of current include current generated by voltage changes in neighboring cells, current applied to the cell from an external source, and the current caused by changes in the cell's voltage. We may then use the equation $I = C_m * dV/dt$ to compute the required voltage change from this current and with that we can find new values for v_x, v_y, and v_z such that they correspond to the new target voltage. We conclude the updates for these variables by applying time-step integration wherever it is dictated by the appropriate HA model specification.

We use the Laplacian operator to calculate the amount of current caused by the potential difference between a given cell and each of its neighbors. Applying this operator yields the sum of the differences between a particular cell and each one of its neighbors divided by the distance between cells squared. We next multiply by the diffusion coefficient of our tissue and by the membrane capacitance C_m to obtain the resulting current. No additional computation is required for the contribution of external stimuli to the total current since its amount, duration, and location are given as parameters to the simulation. The current generated by the cell's voltage change may also be obtained from the equation $I = C_m * dV/dt$.

In the two remaining states, *upstroke* and *plateau & ER*, we only need to update the variables by performing time-step integration. In the next section we will provide a simple simulation algorithm for our HA diffusion models. An important observation is that for each of our HA models we can determine both the initial and final voltages for the *upstroke* and *plateau & ER* states at the time which the state is entered. We will explore this fact in greater detail when we present our idea for an event-driven model in section 1.3.

1.2.1. A Simple HA Diffusion Model Implementation

A straightforward implementation of our HA diffusion model involves computing the necessary updates for each cell for each time-step for the duration of the simulation. We start out by initializing all of our cell variables to reflect the resting potential and set our timer to the start time of our simulation. During each time-step, we check to see if an external stimulus is being applied and update that component of the current accordingly for all affected cells. We proceed to update each variable for every cell depending on the cell's state in the manner previously described. Finally, we increment the timer by one time-step, check to see if output to a file is required, and repeat this process until the timer exceeds the simulation finish time. The C-like code displayed in Fig. 1.2.1 serves to provide a conceptual illustration of this procedure.

```
timer = start_time;
init_variables();
while (timer < end_time)
{
      check_for_external_stimulus();
      for (each cell in simulation)
      {
            if (cell.state == (RESTING or STIMULATED))
                  update_voltage_from_currents(cell);
            update_variables_from_HA_specification(cell);
            check_for_STATE_TRANSITION();
      }
      timer += dt;
      if (output_required(timer))
            OUTPUT_TO_FILE();
}
```

Fig 1.2.1. Straightforward algorithm for simulating HA diffusion models

1.3. An Event Driven Implementation

We can greatly enhance the performance of our HA models by taking advantage of our ability to determine exactly how cells in the *upstroke* and *plateau & ER* states will behave at all times as soon as either one of these states is entered. Given the linear form of our differential equations, our assumption that neighboring cells and external voltages do not play a role in these states, and our knowledge of the values of a cell's variables upon entry into a particular state, we can easily construct explicit equations for each of the variables that take the current time as their only parameters. We may then use these equations to both respond to queries made by neighboring cells (such as those in the *resting & FR* and *stimulated* states) and to determine the time at which a transition to the next state will be required. The latter use for these equations can be performed at runtime by using the Newton-Raphson and solving for the time at which the cell's voltage will be equal to the transition voltage.

Since we can readily generate explicit equations for our cell variables in two of our HA model states, it is evident that direct computation will not be required for cells in these two states during simulation. By calculating the time that these cells will leave these states as soon as the states are entered, we may effectively ignore these cells until their next state changes are required. When the time-step in our simulation is on the order of one microsecond, this can translate into a savings of tens of thousands of computation steps per cell in these states when compared to a simulation using traditional time-step integration. In order to keep track of the next time that each cell requires processing, we propose an event-driven model in which we associate with each cell an event that signifies the type of action required and the time at which it should be performed. We ensure that the correct ordering of events is maintained throughout the simulation by storing them on a priority queue that has been specifically designed to reduce overhead.

In the next couple of sections, we will discuss in greater detail the implementation of such an event-driven HA model along with some techniques we can use to further improve simulation performance. A discussion on the equivalency of our event-driven implementation to the simple implementation presented in section 1.2.1 and performance guarantees of the event-driven model can be found in section 1.4.

1.3.1. Types of Events

We have chosen to classify the events that we use in our model by the type of processing that they require. In a purely event-driven HA diffusion model, it is necessary to have five major types of events, namely the QUERY_NEIGHBOR, Update_State, BEGIN_STIMULATION, END_STIMULATION, and OUTPUT_TO_FILE events.

The QUERY_NEIGHBOR event indicates that its associated cell should update its voltage based upon the amount of external stimulus it is receiving and the effects of its neighboring cells. The way in which it performs these calculations is identical to that of our simple implementation. A QUERY_NEIGHBOR event will continue to generate a new QUERY_NEIGHBOR event for the following time step until the *upstroke* state is reached. When it becomes necessary for the cell to enter the *upstroke* state, we compute how long the cell will be in this new state and an Update_State event is for this calculated time.

The Update_State event indicates the end of the upstroke and plateau states and is handled by performing an action dependent upon the new state of the cell. If this new state is the *plateau & ER* state, we will calculate its exit time and create another Update_State event as we did when we entered the *upstroke* state. Otherwise, if the new state is the *resting & FR* state, we will perform the transition to this state and create a new QUERY_NEIGHBOR event for the succeeding time step.

The remaining three types of events help us to perform actions that are to be applied to the entire simulation as opposed to individual cells. Since the times and locations of external stimuli are known to us at the beginning of the simulation, we can place special events on the queue that will notify us when they are supposed to begin or end. For each cell, we may keep track of the amount of external stimulation being received at any point in time by accumulating these changes in an auxiliary cell variable. The BEGIN_STIMULATION and END_STIMULATION events notify all affected cells when a change in this accumulator should occur. Each cell will then either add or subtract the strength of the stimulus, depending on if the event is a BEGIN_STIMULATION or END_STIMULATION event, respectively. The final event, OUTPUT_TO_FILE, simply writes the voltages of each cell to an output file and generates a new OUTPUT_TO_FILE event for a determined number of time steps into the future.

1.3.2. Handling Regions with Low Activity

In some simulations, it might be possible that large regions of our sample tissue might be at the resting potential for a significant number of time steps with no external stimulation. Since the event-driven nature of our model makes it possible to avoid handling certain cells during a particular time step, as we do for cells in the *upstroke* and *plateau & ER* states, we may extend this idea so that we ignore cells completely at rest until we deem further processing necessary. These cells can easily be "put to sleep" by not generating new QUERY_NEIGHBOR events for them once their voltage drops below a threshold that is sufficiently close to the resting potential. In a fashion similar to how we use explicit equations to respond to queries made to cells in the *upstroke* and *plateau & ER* states, all queries made to sleeping cells will automatically return the resting potential.

Once a cell is considered to be sleeping, there are two ways in which it may be awakened. First, if the voltage of one of its neighboring cells rises above a threshold value, it will be that cell's responsibility to notify all of its neighbors that they must wake up. The second reason a cell should be awakened is if it is affected by an external stimulus through a BEGIN_STIMULATION event. Through this scheme we ensure that no cell remains asleep at a time that its voltage should be above the resting potential while obtaining the benefit of a substantial speedup from skipping over these cells for prolonged periods of time.

1.3.3. Priority Queue Implementation

Because we need to ensure that our events are always executed in the correct chronological order, it is necessary for us to use a priority queue data structure that keeps the events with the lowest times as the highest priority. A standard min-heap is one of the most efficient known data structures for preserving this property. Unfortunately, with insertions and removals costing O($\log_2$ n) time for a min-heap of size n, the overhead cost of simply determining which event should be processed next can in some cases become greater than the time we save by using an event-driven model. This problem is capable of rendering our event-driven approach to be more harmful than good for certain simulation parameters.

When we observe the behavior of our min-heap with respect to the types of events we place on it, it is notable that there are a large number of events that require processing at either the "current" time step (i.e., at the lowest time appearing on the priority queue) or the subsequent time step. The vast majority of these events are QUERY_NEIGHBOR events because in handling this type of event the normal action is to create another event of the same type for a single time step into the future. With a standard min-heap, a great amount of time is spent during the insertion of new QUERY_NEIGHBOR events by placing the new event at the bottom of the heap and swapping it up past all of the events that won't need processing for several thousands of time steps. A similar situation arises when we remove a QUERY_NEIGHBOR event from the top of the heap and perform a fix-heap operation. Our solution is to separate the priority queue into two parts, where one part handles only QUERY_NEIGHBOR events and the other part handles every other type of event.

Since each QUERY_NEIGHBOR event will either occur at the current time or the next time step, we can store a particular QUERY_NEIGHBOR event on one of two lists, where each list reflects one of the two possibilities for event times. When the list for the current time is empty we can swap the two lists continue handling the events for the next time step. The advantage of using lists as opposed to a min-heap in these situations is that insertion and removal on lists can be achieved in constant time. For each type of event other than the QUERY_NEIGHBOR event, we still must use a min-heap to ensure that the ordering of the events is properly maintained. Whenever we need to find the next event for our simulation, we can check to see if there are any events on the min-heap that require processing for the current time. If there are, we will obtain our next event from the min-heap; otherwise, we will continue to consume the events from the lists. The process of selecting from which data structure we should obtain the next event can also be performed in constant time.

By using this implementation we can significantly reduce the overhead caused by our need to maintain an accurate ordering of events. In section 1.4.1.1 we will argue that, by using this type of priority queue instead of a standard min-heap, we can guarantee that for any reasonable simulation our event-driven model will perform no worse than the simple time-step integration model we presented in section 1.2.1.

1.3.4. Initialization and Simulation Loop

The initialization phase for our event-driven model requires that we set the voltages of each cell to the resting potential and place the necessary events onto the priority queue. Because all cells are initially at the resting potential, we may assume that they are all asleep and thus we will not need any QUERY_NEIGHBOR or Update_State events on the queue at this time. We next generate the appropriate BEGIN_STIMULATION and END_STIMULATION events by using the information about the times and locations of external stimuli provided by the simulation parameters. The last event we place on the initial queue is an OUTPUT_TO_FILE event for the time at which the set of output data will be required. At this time we set our timer equal to the start of the simulation, delete all events that occur before the start of simulation, and begin the execution of the main simulation loop.

Instead of incrementing our timer by a single time step after each iteration of the simulation loop, as we did with our simple model in Sect. 3.1, we advance the time in our event-driven model by setting the timer to the time at which the next event on the queue requires processing. We take this event off of the queue, handle it according to what type of event it is, and repeat this process until the time of the next event on the queue exceeds the simulation's finish time. Once we have reached a time greater than the finish time we may terminate the simulation.

The following C-like code in Fig. 1.3.1 provides a high-level description how to carry out an event-driven simulation. Fig. 1.3.2 illustrates a snapshot in time of a hypothetical situation that might arise during a simulation using our event-driven model.

```
timer = start_time;
while (queue.peek_next_event_time() < timer)
      queue.dequeue();
timer = queue.peek_next_event_time();

init_variables();
add_stimulation_events_to_queue();
add_output_event_to_queue();

while (timer < end_time)
{
      event = queue.dequeue();
      handle_event(event);
      timer = queue.peek_next_event_time();
}
```

Fig 1.3.1. Event-Driven algorithm for simulating HA diffusion models

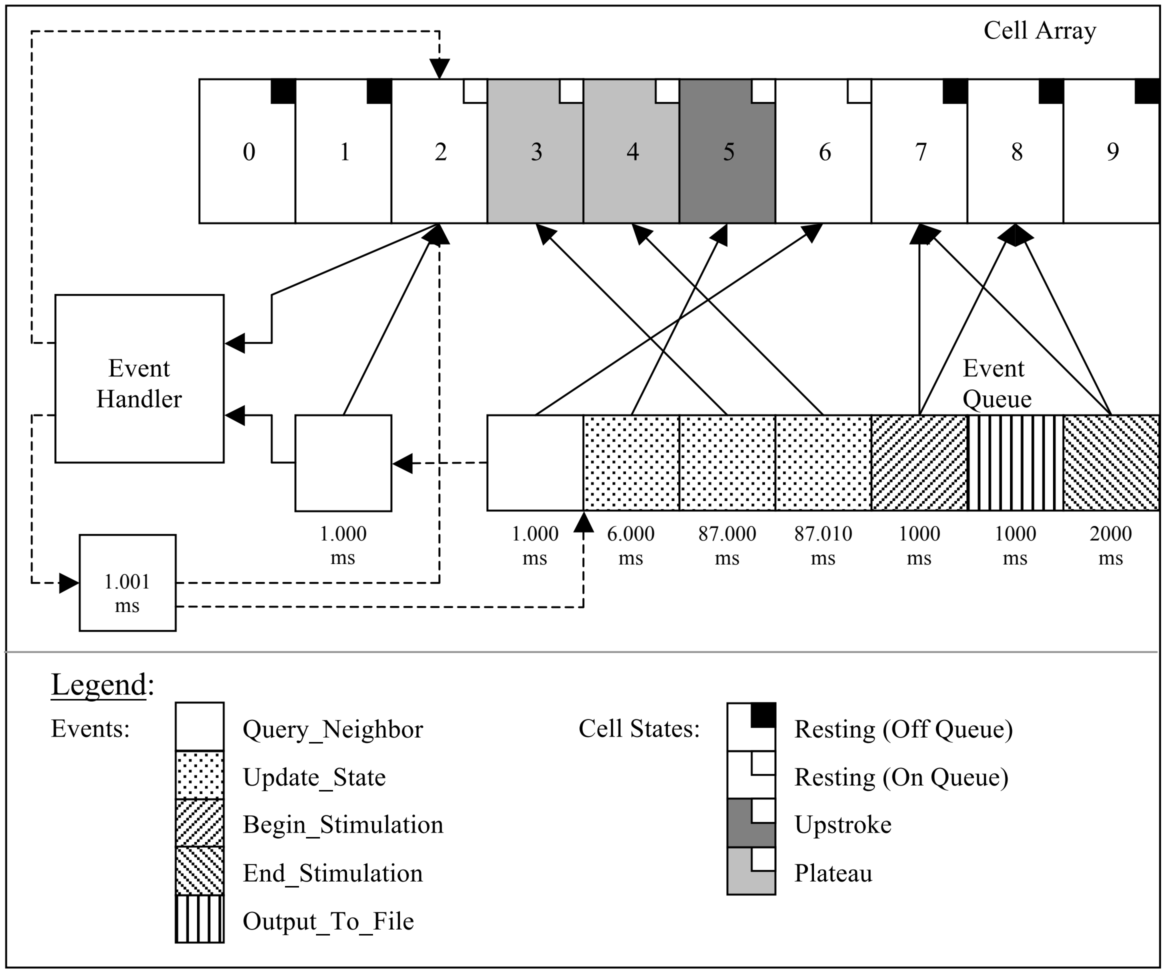


Fig 1.3.2. Simulation Snapshot using the Event-Driven Model

The diagram above depicts a hypothetical situation that might arise when a wave is propagating from left to right. The actual implementation of our Event Queue is significantly more complex than shown in this conceptual diagram. The Event Handler always takes the first event in the queue along with any associated cell(s) as input and can produce a new event and update cell variables as output. The event being processed above is a Query Neighbors event for cell #2 at time = 1.000 ms. The Simulation Manager takes this event off of the queue, updates the appropriate cell #2 variables, determines that cell #2 should remain in the resting state, and produces a new QUERY_NEIGHBOR event for cell #2 for time = 1.001 ms that gets placed into the appropriate position on the queue.

1.4. Comparison of Simulators

In the following sections we will defend our claim that our event-driven model presented in section 1.3 is in fact superior to the simple time-step integration model discussed in section 1.2.1. We will first prove that our event-driven model produces the same results as the time-step integration model does. We will then argue that for all reasonably-sized simulations the event-driven model will outperform the simpler model.

1.4.1. Equivalence of Models

Careful analysis of our event-driven model will show that our model is actually performing all of the same necessary actions as the time-step integration model; the only difference between the two is that our event-driven model skips over certain actions that may be accomplished through more efficient means. The three significant differences between the two models are how we determine the time at which *upstroke* to *plateau & ER* and *plateau & ER* to *resting & FR* transitions occur, how we respond to queries about a cell's voltage during the *upstroke* and *plateau & ER* states, and how we handle cells that are at the resting potential. We will show that these three differences between the two models do not affect the results and as a consequence the system will have the same behavior for each of the states regardless of which model we use.

The first column of the following table summarizes how each model handles cells in each type of situations, namely cells at resting potential, cells above resting potential in the *resting & FR* and *stimulated* states, and cells in the *upstroke* and *plateau & ER* states. The second column lists the possible sources for error associated with the approach each model takes.

Model and Situation	**Required Cell Processing**	**Associated Error**
Resting Potential		
Time-step Integration	-Handle cells as if they were any other cell in the Resting & FR state -Performing normal cell updates will result in this cell remaining at resting potential	-No error with respect to a diffusion model where only neighboring cells affect voltage
Event-Driven	-Leave cells off of the queue until a change occurs in a neighboring cell or if an external stimulus is applied -Respond to queries by returning resting potential	-No error with respect to a diffusion model where only neighboring cells affect voltage

Resting & FR/Stimulated		
Time-step Integration	-Update cell variables and check the affects of neighbors and external stimuli -Respond to queries by returning the cell's voltage from the previous time step	-Approximation error caused by using time-step integration instead of using explicit solutions to differential equations -Truncation of state transition time to nearest whole time step
Event-Driven	-Generate an event every time step that performs the same updates for the cell as with the TSI model -Respond to queries by returning the cell's voltage from the previous time step	-Approximation error caused by using time-step integration instead of using explicit solutions to differential equations -Truncation of state transition time to nearest whole time step
Upstroke/Plateau & ER		
Time-step Integration	-Update cell variables based solely upon differential equations using time-step integration -Respond to queries by returning the cell's voltage from the previous time step	-Approximation error caused by using time-step integration instead of using explicit solutions to differential equations -Truncation of state transition time to nearest whole time step
Event-Driven	-Solve for explicit solution of differential equations and use Newton-Raphson to calculate the next transition time -Respond to queries by evaluating explicit equations for the time of the previous time step	-No error with respect to using explicit solutions to differential equations -Truncation of state transition time to nearest whole time step

From this table, it is evident that in all situations our event-driven model either has the same error associated with a time-step integration model or no error at all with respect to the actual solutions for our differential equations. We will conclude this discussion with an elaboration on why in our event-driven model sleeping cells will never fail to be awakened at times when their voltages should be above resting potential and why we may always compute the time a cell will leave the *upstroke* or *plateau & ER* states to an accuracy within the nearest time step.

1.4.1.1. Determining the Appropriate Time for Awakening Sleeping Cells

In the case where we have cells remaining at resting potential for prolonged periods of time in the simpler model, the cells in these regions are effectively doing nothing other than keeping all of their variables unchanged. In the event-driven model, by putting cells to sleep, we are actually performing the exact same action. The only concern is that our event-driven could potentially fail to immediately awaken cells as soon as their voltages should begin to change. To prove that this will not be an issue, we will look back to our discussion in section 1.2 on the three types of current that induce changes in voltage and show that the event-driven model correctly handles each type of current.

There is no change in the voltages of cells at resting potential because they will remain at this potential until they are otherwise stimulated, meaning the current caused by voltage change should have no effect on sleeping cells. When there is an external stimulus applied to a cell at resting potential, it should begin taking this into consideration and resume querying its neighbors and checking to see if a state change is necessary. This action is reflected in our event-driven model by making a BEGIN_STIMULATION event force each sleeping cell that it affects to wake up and begin generating new QUERY_NEIGHBOR events. The final source of current, caused by potential differences between neighbors, takes time to propagate in the time-step integration model. This is due to the fact that differences in voltage can only affect a cell's immediate neighbors during a single time step. Our event-driven model captures this notion by obligating any cell noticing a voltage increase above a threshold at which neighbors may be affected to notify its sleeping neighbors that they should awaken and begin generating

QUERY_NEIGHBOR events for the next time step. Because all of the currents are correctly taken into account at all times, our event-driven model will produce the same results for cells at resting potentials while doing significantly less work when compared time-step integration models, including models that make use of adaptive time steps.

1.4.1.2. Accurately Computing Transition Times Using Explicit Equations

Since the differential equations for cell variables in the *upstroke* and *plateau & ER* states are linear, they each have a general solution in the form of an exponential equation $ce^{(\alpha(t-t_0)}$, where α is the coefficient from the linear differential equation, t_0 is the time at which the state was entered, and c is the value of the variable upon entry into the state. When the voltage of a cell is defined as the sum of these cell variables, as it is the case with our HA model specifications, the explicit equation for the voltage in these states is the sum of equations of this general form.

Because we already know the value of the voltage that will trigger a state transition from the HA model specification, we may set this equation equal to this value and solve for t, the only unknown, to find the time at which the transition will occur. We may use the Newton-Raphson method to obtain a value for t which is guaranteed to be within a desired accuracy, which in our case will be to the nearest full time step. Once we know the value of t, we may use the explicit equations for each cell variable to determine their values for the time at which the state will be exited.

The only source of error in these transition times will be due to the truncation of all event times to the nearest whole time step. A similar problem is present while using time-step integration techniques because all updates are performed on whole number time steps and there are no means to handle transitions at times in between time steps. Thus, our calculation of the state transition time in the event-driven model will no less accurate than a transition time obtained from the simpler model.

1.4.2. Performance Guarantees

At first glance, one might think that the use of a priority-queue in our event-driven model could potentially be the source of enough overhead to negate the amount of computation time saved by using explicit equations for cells in the *upstroke* and *plateau & ER* states. However, when we use the specialized priority queue introduced in section 1.3.3, we can prove that our event-driven model will not be outperformed by a time-step integration model by more than a small constant factor for any reasonable simulation. We define a reasonable simulation to be any simulation containing a small enough number of cells such that all of the cell variables may be stored in the memory system of a modern computer.

To carry out our performance analysis, we must first define a basic unit of work. One logical choice would be the amount of work required to update a singe cell's variables during a single time step in a standard time-step integration model. To help simplify our proof, we will also assume that the amount of time it takes to swap an element up or down by one level in a fix-heap operation is also equal to a single unit of work. This assumption is a conservative one because the amount of work done in calculating new cell variable values is significantly more than that performed in a single fix-heap swap operation; by allowing this simplification we are actually overestimating the amount of overhead caused by our queue.

We compare the amount of work per cell per time step required for each situation in each model in terms of this defined unit of work in Table 1.4.1. In addition to the time-step integration and event-driven models we will also consider an adaptive time-step model, which is a modified version of the time-step integration model in which the computation for cells at resting potential is skipped for a predetermined number of time steps. Table 1.4.2 lists typical values for some of the variables introduced in Table 1.4.1.

(Table 1.4.1)

Model	Resting Potential	Resting & FR/Stimulated	Upstroke/Plateau
TSIM	1	1	1
ATSIM	α	1	1
EDM	0	$1 + \beta$	$((\log_2 n) + c)/s$

(Table 1.4.2)

Variable	α	β	c	n	s
Value	$0< \alpha <1$	$0< \beta <1$	$0 < c < 20$	Varies	> 1000*

*- typical value when assuming a time-step on the order of one microsecond.

The variable α represents the amount of work required to determine that a cell should be skipped over for a time-step while β represents the amount of overhead required to remove the next QUERY_NEIGHBOR event from a linked list. The variable c indicates the amount of processing a cell requires for computing its transition time using Newton-Raphson and s is the minimum number of time steps spent in any state that makes use of explicit equations, to be conservative. The final variable n is the maximum number of events that may be on the min-heap part of the queue at any given time. We will discuss in greater detail the derivation of the expression $((\log_2 n) + c)/s$ in section 1.4.2.1 after arguing that in any realistic simulation setting its value will be no greater than one.

Solving for the minimum number of events that must be on the queue to require more than a single unit of work per cell per time step yields the expression $n = 2^{(s-c)}$. It is evident from the typical values of s and c that we would need to have a much larger number of events than we could possibly store in memory in order to have the event-driven model perform worse than the other two models in the *upstroke* and *plateau & ER* states. It thus follows from Table 1.4.1 that in an absolute worst-case scenario our event-driven model will not perform worse than the other two models by any more than a factor of $1 + \beta$ because for all reasonable simulations the other two models are at best capable of outperforming the event-driven model in one type of situation by this factor.

1.4.2.1. Complexity Analysis for Overhead Caused By Min-Heap

Because it is difficult to directly assess how much overhead a single event on the min-heap component of our queue generates per time step, we will instead investigate the amount of overhead a cell is responsible for throughout the duration of time it spends on the min-heap. Supposing that we have O(n) events on the min-heap at any particular time, it is reasonable to assume that while an event is on the min-heap we will have O(n) insert and remove operations performed (likely one insertion and one removal for each other event). Because each of these operations requires $O(\log_2 n)$ units of work to complete, the total amount of work done while an event is on the min-heap will be

$O(n \log_2 n)$ units. We can use the amortized argument that each cell is thus responsible for $O(\log_2 n)$ of these units because each associated event on the min-heap contributes equally to the total overhead.

For the entire time a cell spends in either the *upstroke* state or *plateau & ER* state, the total amount of work required will be the sum of c, the number of units required to compute the exit time from the state, and the amount of queue overhead, which we have computed to be $O(\log_2 n)$. To determine how much upkeep is required per time step for this cell, we divide the amount of total work by s, the number of time steps that the cell spends in this state. Using this information, we may define an expression for the units of work needed per cell per time step to be $(O(\log_2 n)+c)/s$ for the *upstroke* and *plateau & ER* states.

1.5. Simulation Results

We compared the time required to produce a complete spiral, from the second stimulus time onwards, with a time-step integration simulator and with an event-driven simulator for a 400-x-400 array of EHAs for the Luo-Rudy nonlinear model. The event-driven simulator completed the 500 ms simulation in 1.70 hours as opposed to 9.05 hours for the time-step integration simulator, showing more than a five-fold speedup for the event-driven simulator.

We also ran simulations in which we applied a stimulus to the corner of different-sized EHA arrays. The simulations ran until the last EHA returned to the resting state. The timing results for both simulators resulted in an average speedup of 1.25 for the event-driven simulator. This is an example of a simulation in which one might expect the event-driven simulator to not achieve top performance due to the large number of cells in non-refractory states.

2. Simulator Features

2.1. Individual Cell Reports

Both simulators provide a feature for producing output reports for specified cells during the course of the simulation. At every standard output interval, an individual cell report will include the simulation time and voltage value for the specified cell.

Each two column report file can then be imported by plotting software, such as MATLAB, so that the morphology of each of the cell's action potentials can be viewed graphically. The cells for which the reports are generated and the filenames of the cell reports are specified by the cell report input file (see sections 3.3.2.1.3 and 3.3.2.2.3).

2.2. Graphical Simulation Snapshots

Both simulators have functionality included for producing graphical snapshots of the simulation to show wave propagation. The output files show the cell grid colored by the state of each cell (white for upstroke, gray for plateau, and black for resting and stimulated) in .ppm format. These graphical files are produced at regular intervals (see section 3.3.4) and can be viewed with programs such as Irfan View and Advanced Batch Converter.

2.3. Customizable HA Parameters

It is possible for each simulator to change the values of HA constants such as the coefficients for the differential equations, the voltage values for each transition, and the resting potential. This can be done by specifying an input file containing this information through a command-line argument (see sections 3.3.2.1.2 and 3.3.2.2.2). The user also has the option of using the default values for these parameters by not providing an HA specification file.

2.4. Load/Save Simulation State Functionality (Event Driven Simulator)

The event-driven simulator includes a mechanism for saving and loading simulation states. The saving state feature allows for state files to be created at times specified by a state saving information file (see section 0). The loading state feature allows one of these state files to be loaded along with a new stimulus input file. The benefit of this feature is that simulations can be resumed from specified points with different stimuli, eliminating the need to repeat the earlier part of the simulation.

3. Simulator Instruction Manual

3.1. Environment Information

Both simulators were originally developed on the UNIX compserv1.cs.sunysb.edu using the standard gcc compiler with the –lm commands enabled (for using math commands). The simulators were later ported to a standard Windows machine using a free C/C++ development environment called Dev-C++ which can be downloaded from www.bloodshed.net.

The reason for porting the simulators to the Windows machine was to obtain accurate simulation timing results. The compserv1.cs.sunysb.edu machine is a shared user machine, so the timing results tended to vary greatly from one run to another as the result of the other users' activity. Running simulations on a dedicated, single-user machine is the only way to guarantee accurate timing results.

There are a few subtleties between the two compilers involving the constant definitions of true, false, and null (Dev-C++ has these predefined while gcc does not). The code in its current state is configured to run with Dev-C++.

3.2. Setup Procedures

All code files for each simulator must be placed in the same directory before the compiler is run. All filenames used in the simulators are relative to the directory containing the executable files. The executable file assumes that the input file is called testdata.txt and is located in this directory unless a command-line parameter specifies otherwise. The output file will be placed in this directory and will be named AP.dat unless a command-line parameter specifies otherwise. The log file with the timing results will be placed in this directory and will be named log.txt unless a command-line parameter specifies otherwise.

3.3. Simulation Input and Usage

Both simulators share a common framework for usage and how input information is collected. Command-line parameters are used to specify which input files are going to be used and the file paths that they correspond to. Input files provide information to the simulators such as the general simulation parameters, the stimulus information, the HA specification constants, the list of cells to generate reports for, and a list of times to create saved state files (for the event-driven simulator). Preprocessor options allow certain features to be compiled out of the simulators to reduce overhead during simulation timing runs. Both simulators contain a built-in mechanism for producing graphical snapshots of the simulation at regular intervals.

The next sections detail how each part of this framework can be used to customize the simulation to meet the needs of a particular experiment. Examples are provided to ensure the clarity of the usage descriptions.

3.3.1. Command-Line Parameters

Command-line parameters are provided to the simulators at runtime and consist of a parameter identifier followed by a parameter target. A parameter identifier consists of a hyphen followed by one or two letters, such as '-sf' or '-a', which signifies the option that the following target belongs to. The parameter target is usually the name of a file path that can be at most 40 characters in length. Any number of command-line options may be selected by listing the identifiers and targets one after another. The following example shows three options selected along with three target paths specified:

```
"-i Input\input.txt –o Output\output.txt –r Input\reportFile.txt"
```

The following is a list of all of the command-line parameters that can be passed to the simulators. With the exception of the '-sf' and '-sl' identifiers, which are particular to the event-driven simulator, each command-line parameter may be used for event-driven and time-step integration simulators.

- `-a <target>`: Indicates that the HA specification constants are to be read from an input file found at location `target`. See sections 3.3.2.1.2 and 3.3.2.2.2 for the format of these files. Omitting this parameter will use the default values for all HA constants.

- `-i <target>`: Indicates that the location of the general parameter and stimulus file will be `target` instead of testdata.txt. Omitting this parameter will use the file located at testdata.txt.

- `-o <target>`: Indicates that the location of the output file will be `target` instead of AP.dat. Omitting this parameter will use the file located at AP.dat.

- `-l <target>`: Indicates that the location of the log file (with simulation timing results) will be `target` instead of log.txt. Omitting this parameter will use the file located at log.txt.

- `-r <target>`: Indicates that individual cell reports will be generated based upon the individual cell report file located at `target`. The cell report preprocessor option must be turned on for the reports to be generated (see section 3.3.3). See sections 3.3.2.1.3 and 3.3.2.2.3 for the format of these files. Omitting this parameter will result in no cell reports being generated.

- `-sf <target>`: Indicates that save state files will be generated for the event-driven simulator. The file containing information about the times at which these files should be created is located at `target`. See section 2.4 0 for the format of this file. Omitting this parameter will result in no saved states being created.

- `-sl <target>`: Indicates that simulation will be started for the event-driven simulator from the state represented by the saved state file located at `target`. Omitting this parameter will result in the simulation starting at the specified start time with all cells at resting potential.

3.3.2. Input File Format

3.3.2.1. Event Driven Simulator

The event-driven simulator can accept up to four different types of input files to be used during a simulation. The following sections describe in detail the formats of these files.

3.3.2.1.1. General Parameter and Stimulus File

The General Parameter and Stimulus File provides all of the basic information to the event-driven simulator about simulation parameters and each stimulus. The general format of this input file is as follows:

```
<sz> <sti> <dd> <C> <dif> <st> <end> <dt> <out>
<begin> <en> <up> <low> <left> <right> <stren>
       (Repeat Last Line for Each Stimulus)
```

The abbreviations listed above represent the following information:

- `sz`: The size of the simulation grid. A value of n indicates that the simulation will use an n x n grid.
- `sti`: The number of stimuli to be used in the simulation. This number represents the number of lines that will follow the first one in the input file, with each representing a different stimulus.
- `dd`: The physical distance between two neighboring cells.
- `C`: The membrane capacitance of the cell.
- `diff`: The diffusion coefficient for the simulation.
- `st`: The starting time of the simulation, given in milliseconds.
- `end`: The finish time of the simulation, given in milliseconds.
- `dt`: The size of the simulation time-step, given in milliseconds.
- `out`: The number of iterations (time steps) between each voltage dump to the main output file. The output interval is effectively `dt * out` in milliseconds.
- `begin`: The start time of the stimulus, given in milliseconds.
- `en`: The end time of the stimulus, given in milliseconds.
- `up`: The upper coordinate of the stimulus bounding box, with lower numbered coordinates being further upward.
- `low`: The lower coordinate of the stimulus bounding box, with higher numbered coordinates being further downward.

- `left`: The left coordinate of the stimulus bounding box, with lower numbered coordinates being further left.
- `right`: The right coordinate of the stimulus bounding box, with higher numbered coordinates being further right.
- `stren`: The strength of the stimulus, given in amps.

The following is a sample General and Stimulus File with three different stimuli:

```
100 3 0.2 1 0.4 0 100 0.001 1000
0 1 0 19 0 19 -0.60
60 61 80 99 80 99 -0.60
30 31 40 59 40 59 -0.60
```

The above input file specifies a simulation that lasts 100 milliseconds with a single microsecond time-step. Output occurs at the end of every simulation millisecond. Each of the three stimuli lasts for a single millisecond and occurs at a different time and in a different region of the 100 by 100 simulation grid.

3.3.2.1.2. Hybrid Automata Specification File

The Hybrid Automat Specification File provides a mechanism for overriding the event-driven simulator constants that normally describe various coefficients of the corresponding hybrid automata. The general format of this input file is as follows:

```
<ax0> <ay0> <az0>
<ay1> <az1>
<ax2> <ay2> <az2>
<ax3> <ay3> <az3>
```

```
<vO> <vT> <vS> <vR> <vW> <rp>
<initVx> <initVy> <initVz>
<guessUpstroke> <guessPlateau>
```

The abbreviations listed above represent the following information:

- `<a<cur><st>>`: The coefficient for the differential equation in state `st` for the voltage designated by the letter `cur` (x, y, or z).
- `<vO>`: The overshoot potential given in millivolts above resting potential.
- `<vT>`: The threshold potential (for starting an upstroke) given in millivolts above resting potential.
- `<vS>`: The stimulus potential (for detecting a stimulus event) given in millivolts above resting potential.
- `<vR>`: The repolarization potential (for the end of the absolute refractory period) given in millivolts above resting potential.
- `<vW>`: The potential for waking cells and putting them to sleep given in volts.
- `<rp>`: The resting potential given in volts.
- `<initV<cur>>`: The initial value for the voltage designated by the letter `cur` (x, y, or z), given in millivolts above resting potential.
- `<guessUpstroke>`: The estimated length of the upstroke phase in milliseconds used to assist Newton's method.
- `<guessPlateau>`: The estimated length of the plateau phase in milliseconds used to assist Newton's method.

The following is a sample Hybrid Automat Specification File for the event-driven simulator with parameters equal to the system defaults (the NNR model):

```
-0.025 -0.07 -0.2
-0.07 -0.2
250 200 125
-0.025 -0.07 -0.2
120.0 30.0 0.0005 20.0 -0.0799 -0.08
0 0 0
0.01 78.2
```

3.3.2.1.3. Individual Cell Report Information File

The Individual Cell Report Information File specifies which cells should have their voltages reported (for plotting purposes) and what the corresponding report file name should be. The general format of this input file is as follows:

```
<reportCount>
<cellX> <cellY>          //Repeat these two lines
<filename>               //for each cell reported
```

The abbreviations listed above represent the following information:

- `<reportCount>`: The number of cells that will generate reports. The number of lines following this one will be equal to `2 * reportCount`.
- `<cellX>`: The x coordinate of the cell being reported in the file specified by the following line.
- `<cellY>`: The y coordinate of the cell being reported in the file specified by the following line.
- `<filename>`: The file name of the report corresponding to the cell listed on the previous line.

It is important to note that this feature will only work if the CELL_REPORTS_ON preprocessor directive is enabled (see section 3.3.3). Also, if the cell coordinates are out of bounds for one of the entries, no report will be generated for that cell.

The following is a sample Hybrid Individual Cell Report Information File for the event-driven simulator:

```
4
0 1
cell0001.dat
10 10
cell1010.dat
8 7
cell0807.dat
-1 4
invalid.dat
```

The previous example lists 4 different cell report specifications. However, only 3 are valid (assuming a grid size of greater than 10) because -1 is out of range. Reports for (0,1), (10,10), and (8,7) are created and stored in the files cell0001.dat, cell1010.dat, and cell0807.dat, respectively.

3.3.2.1.4. State Saving Information File

The State Saving Information File specifies at which times the event-driven model should dump its state to a file and the file name that each dump time corresponds to. The general format of this input file is as follows:

```
<stateCount>
<dumpTime>                 //Repeat these two lines
<filename>                 //for each state dumped
```

The abbreviations listed above represent the following information:

- `<stateCount>`: The number of states to be saved. The number of lines following this one will be equal to `2 * stateCount`.
- `<dumpTime>`: The simulation time, in milliseconds, at which the state should be dumped to the file specified by the following line.
- `<filename>`: The saved state file corresponding to the dump time specified by the previous line.

It is important to note that if the dump time is out of the simulation time bounds for one of the entries, no saved state file will be generated for that dump time.

The following is a sample State Saving Information File for the event-driven simulator:

```
3
1.0
state1.dat
2.0
state2.dat
-1.0
invalid.dat
```

The previous example lists 3 different saved state specifications. However, only 2 are valid (assuming a simulation start time of 0.0 and end time greater than 2.0) because -1.0 is out of range. Saved states for times 1.0 and 2.0 milliseconds are created and stored in the files state1.dat and state2.dat, respectively.

3.3.2.1.5. Saved State File

While Saved State Files are generally generated by the event-driven simulator, it might be useful at times to manually alter the starting state for simulations by modifying this type of file. The general format is described by the following grammar in BNF-like notation followed by a more detailed description of `cell_info` and `event_info`:

```
<saved_file> :=     <cell_part>
                    <queue_part>
<cell_part> :=      (#_of_cells)
                    <cell_info>    //Cell (0,0)
                    <cell_info>    //Cell (0,1)
                       ...         //Repeat
<queue_part> :=     <list_part>    //Now List
                    <list_part>    //Later List
                    <heap_part>
<list_part> :=      (capacity) (size) (list_time)
                    <event_info>   //1st event
                    <event_info>   //2nd event
                       ...         //Repeat
<heap_part> :=      (capacity) (size)
                    //Heap in Array Notation
                    <event_info>   //1st event
                    <event_info>   //2nd event
                       ...         //Repeat
```

The token `cell_info` represents the state information for a particular cell. It consists of the following 18 fields on a single line, each separated by a single space, in the order shown below:

- State code of the cell
- X position of the cell
- Y position of the cell
- The last time-step's value of Vx
- The current time-step's value of Vx
- The last time-step's value of Vy
- The current time-step's value of Vy
- The last time-step's value of Vz
- The current time-step's value of Vz
- The last time-step's value of V
- The current time-step's value of V
- The value of Vn
- The last time-step's time
- The current time-step's time
- The value of time_enter (used in upstroke and plateau states)
- The value of time_exit (used in upstroke and plateau states)
- The amount of applied stimulus (in A)
- A flag for whether or not the cell is awake

The token `event_info` represents the state information for a particular event. It consists of the following 4 fields on a single line, each separated by a single space, in the order shown below:

- X position of the affected cell
- Y position of the affected cell
- The type code of the event
- The time of the event

3.3.2.2. Time-step Integration Simulator

The time-step integration simulator can accept the same types of input files as the event-driven simulator with the exceptions of the state saving information file and the saved state file. The following sections describe in detail the formats of these files by describing how they differ from the file formats in the event-driven model.

3.3.2.2.1. General Parameter and Stimulus File

The General Parameter and Stimulus File for the time-step integration simulator is effectively the same as that for the event-driven simulator with a couple of additional restrictions. The list of stimuli must be listed in chronological order in the input file to avoid skipping over certain stimuli. The time-step integration simulator also is not capable of handling two different stimuli at the same time. These restrictions are not placed upon the event-driven model because it is powerful enough to handle these cases without the more sophisticated (and time-consuming) logic that the time-step integration model would require.

3.3.2.2.2. Hybrid Automata Specification File

The Hybrid Automat Specification File for the time-step integration simulator is similar to the one for the event-driven simulator with the omission of several parameters. The general format of this input file is as follows:

```
<ax0> <ay0> <az0>
<ay1> <az1>
<ax2> <ay2> <az2>
<ax3> <ay3> <az3>
<vO> <vT> <vS> <vR> <rp>
<initVx> <initVy> <initVz>
```

The abbreviations listed above have the same meanings as those listed in section 3.3.2.1.2. The upstroke and plateau phase length approximations are not required because the time-step integration simulator does not use Newton's method. The parameter `vW` is also not required because the time-step integration simulator does not put cells to sleep.

The following is a sample Hybrid Automat Specification File for the time-step integration simulator with parameters equal to the system defaults (the NNR model):

```
-0.025 -0.07 -0.2
-0.07 -0.2
250 200 125
-0.025 -0.07 -0.2
120.0 30.0 0.0005 20.0 -0.08
0 0 0
```

3.3.2.2.3. Individual Cell Report Information File

The Individual Cell Report Information File format for the time-step integration simulator is identical to that of the event-driven simulator. See section 3.3.2.1.3 for more details.

3.3.3. Preprocessor Options

Since the simulation runs for producing timing results require optimal performance, it is possible to toggle compile time flags for both simulators to disable additional functionality not required for such runs (individual cell reports, graphical output, etc.).

The following list describes all features that may be enabled or disabled and details which source files they may be found in:

- **STATE_SNAPSHOTS_ENABLED**: This enables (1) or disables (0) whether or not graphical output will be produced by the simulators. This constant is located in the main.c file of both simulators.

- **ALLOW_CELLS_TO_SLEEP**: This enables (1) or disables (0) whether or not the event-driven simulator puts cells at resting potential to sleep. This constant is located in the cell.h constant in the event-driven simulator only. Disabling this option for simulations with many stimuli (and thus fewer cells at resting potential) can improve the performance of the event-driven simulator.

- **CELL_REPORTS ENABLED**: This enables (1) or disables (0) whether or not individual cell reports will be produced by the simulators. This constant is located in the param.h file of both simulators.

3.3.4. Graphical Simulation Snapshots

The mechanism for creating graphical snapshots of simulations for both simulators requires that a directory be set up at the path Output\Dump relative to the location of the simulator executable file. The frequency of snapshot creation is set by the constant STATE_SNAPSHOTS_INTERVAL defined in main.c for both simulators. This constant represents the number of standard voltage dumps that occur between the creation of consecutive snapshot files.

This feature is only enabled if the preprocessor option STATE_SNAPSHOTS_ENABLED is turned on (see section 3.3.3). The generated snapshot files are located in the Output\Dump directory (relative to the location of the simulator executable file) and each have the name snap*xxxx*.ppm, where *xxxx* is the number of standard voltage dumps that occurred before the creation of the snapshot file. The number *xxxx* includes leading zeroes so that alphabetical ordering is preserved, making it easier for the user to view the snapshots in a slideshow type manner.

4. Code Maintenance Manual

The following sections serve as a reference for maintaining the code of both simulators. These sections detail the various source files for each simulator and highlight where to find the important functions used to carry out simulations.

Each simulator has a series of header (.h) files that define constants and various structs used in the simulator. As virtually all of these structures are well commented in the source code and self-explanatory, the following sections will focus much more on the functional (.c) source files.

4.1. Event Driven Simulator

The event-driven simulator makes use of several source files that separate the core functional units from one another. The files heap.c, list.c, and queue.c define the operations required for maintaining the correct ordering of events. The event.c file provides various operations for manipulating events. The param.c file contains functions for processing the various types input information from the user and setting up the simulation-wide parameter structure. The cell.c file contains operations on cells that are used inside of the main simulation loop. The last file, main.c, contains the main operating function that ties all of the other pieces together. The most important functionality that can be found in each of these files is further detailed in the following subsections.

4.1.1. queue.c

The queue.c file provides operations for the specialized priority queue that is used by the event-driven simulator. The priority queue contains two lists and one heap for storing the different types of events. The queue operations provided include export/importQueue (for state saving and loading), create/destroyQueue, peekNextTime (for obtaining the next event time), and the traditional enqueue and dequeue operations.

The queue works to ensure that the correct ordering of all events is maintained. Because the QUERY_NEIGHBOR events can only be scheduled for the current or next time-step, each of the queue's two lists contains all QUERY_NEIGHBOR events for one of these two possible time-steps. All of the other types of events are placed on the heap. The enqueue determines which of the three data structures should receive the incoming event and performs the insertion using the appropriate operation. The dequeue operation determines which of the two data structures (the heap and the list for the current time-step) has the next event and removes the appropriate event from the correct data structure. The queue attempts to improve the performance of these operations by using flags to designate which data structure contains the next event and whether or not the lists are empty; because QUERY_NEIGHBOR events are by far the most commonly enqueued and dequeued type of events, the queue attempts to offload most of these "queue-state" calculations to operations involving the heap.

4.1.2. list.c

The list.c file provides operations for maintaining a list of events that are to all occur at the same time. It is implemented as a circular array of fixed capacity that may be expanded by calling the expandList operation. This operation is called automatically by the addToTail operation if the list full, although it is recommended that this check be commented out to achieve peak performance during simulations because an upper bound on the list size may be calculated at the start of the simulation. The operation addToTail adds a new event to the end of the list while the removeFromHead operation removes and returns the first event from the list. Both of these operations may be performed in constant time. The create/destroyList operations handle the dynamic memory allocation, and the import/exportList operations handle state saving and loading. Each list has an associated time, which matches the time of all of its events, and this attribute can be retrieved through the getTime operation.

4.1.3. heap.c

The heap.c file provides the operations required for a standard implementation for a min-heap. It is implemented as its array notation of fixed capacity that may be expanded by calling the expandHeap operation. The insert and removeNext operations are the standard heap operations for inserting an arbitrary new event and removing the event with the highest priority, respectively. Both of these operations fix the heap based upon the priorities of the events in O($\log_2$ n) time, where higher priority is defined primarily by a lower event time and secondarily by the type of the event (stimulation events have priority over non-stimulation events with the same time). The getNextTime event reports the time of the event with the highest priority on the heap. The create/destroyHeap operations handle the dynamic memory allocation, and the import/exportHeap operations handle state saving and loading.

4.1.4. event.c

The event.c file provides some very basic operations that may be performed on events. The create/destroyEvent operations handle the dynamic memory allocation, and the import/exportEvent operations handle state saving and loading. The remaining operation, provideStimulusInfo, simply accepts a stimuluation event and populates it with information about the stimulus (location, strength, etc.).

4.1.5. param.c

The param.c file contains all operations relating to the reading of input files and command line parameters. The function processCommandLineArgs parses the command-line input and performs the appropriate action using the specified argument for each command-line parameter. Each of the remaining operations populates the simulation-wide parameter structure with the information provided by the corresponding input files:

- readHACoefficients: Hybrid Automata Specification File (see section 3.3.2.1.2)
- readParam: General Parameter and Stimulus File (see section 3.3.2.1.1)
- readCellReportInfo: Individual Cell Report Information File (see section 3.3.2.1.3)
- readStateInfo: State Saving Information File (see section 3.3.2.1.4)

4.1.6. cell.c

The cell.c file consists of a variety of operations that process cells throughout the course of a simulation. The create/destroyCell operations handle the dynamic memory allocation, and the import/exportCell operations handle state saving and loading. The initializeParamWithDefaults operation initializes the HA coefficients to their default values and the function newton is an implementation of Newton's method used to solve for the times that state transitions will occur.

The remaining functions are specifically called by the main simulation loop throughout the course of the simulation. The functions notifyCell and putCellToSleep are used for waking up sleeping cells and putting cells below the sleeping threshold, respectively. The function getVoltage calculates the voltage for a particular cell at a particular time, regardless of its current state. The function getCurrent approximates the current for cells in the resting and stimulated states (this function is undefined for the upstroke and plateau states).

The updateVoltage function starts by performing time-step integration on a cell by taking into consideration its current, the affects of its neighbors, and applied external stimuli. This function is where the diffusion model equation is implemented. The remainder of the function handles all state transitions from resting to stimulated, stimulated to resting, and stimulated to upstroke as well as the sending of alerts to the main simulation loop when neighboring cells should be awakened and if a transition to the upstroke state is required.

The beginUpstroke function is called when a cell is about to enter the upstroke state. It correctly sets up the cell's variables so that the voltage can be computed at any time while it is in this state and calculates the time that it will be leaving this state. The updateState function actually performs the transitions from upstroke to plateau and from plateau to resting and returns the calculated time for the cell's next transition.

It is important to note that cells maintain their variable values for both the current and previous time-steps. The reason that this is done is because some cells will have time-step integration performed on them before others will, and the old values of the variables are required to emulate the process of updating all cells at the same exact time.

4.1.7. main.c

The file main.c contains the main simulation function along with a couple of helper functions. The functions import/exportCells handle state saving and loading for the entire grid of cells. The function fileStr generates file names for state snapshot files (see section 3.3.4) such that alphabetical ordering of the file names is the same as their chronological ordering. The main function actually carries out the simulation by proceeding through various phases, namely the input reading phase, the initialization phase, the main simulation phase, and the cleanup phase.

The input reading phase begins by processing the command-line parameters and by reading the appropriate input files (see section 4.1.5). The output files are also set up at this point for writing. The initialization phase follows by checking to see if a loading from a state file is required. If loading is required, the importCells and importQueue operations are called to retrieve the previous simulation state (with the exception of the original stimulation events). Otherwise, all of the cells are created at the resting state and the initial events are placed onto an initially empty queue. In either case, the appropriate stimulation events are added to the queue. This mechanism allows the times and locations of stimuli to be changed from one simulation to another, even if the two simulations use the same initial state.

The main simulation loop will repeatedly take the next event off of the queue and process it until the time of the next event exceeds the ending time of the simulation. Each event is handled as was described in section 1.3.1. The only type of event that performs any additional actions is the OUTPUT_TO_FILE event, which handles a few other output related tasks such as generating cell reports and creating simulation snapshot files. The SAVE_SYSTEM_STATE event is used to save the current simulation state to an output file as specified by the State Saving Information File (see section 3.3.2.1.4).

The final phase of the main simulation procedure is the cleanup phase. This phase consists of a final output dump, the deallocation of simulation resources, and the closing of all output file pointers.

4.2. Time-step Integration Simulator

The time-step integration simulator is essentially a simplified version of the event-driven simulator. The files event.c, heap.c, list.c, and queue.c are not part of the time-step integration simulator because the simulation process makes no use of events. The following subsections detail any significant differences in the time-step integration simulator files with respect to the event-driven simulator.

4.2.1. param.c

The param.c file for the time-step integration simulator is very similar to the one found in the event-driven simulator, except that it doesn't contain a function for reading state info and the format of the input files is slightly different. The following is a list of the functions for reading the input files with references to the file format:

- readHACoefficients: Hybrid Automata Specification File (see section 3.3.2.2.2)
- readParam: General Parameter and Stimulus File (see section 3.3.2.2.1)
- readCellReportInfo: Individual Cell Report Information File (see section 3.3.2.2.3)

4.2.2. cell.c

The differences between the cell.c file found in the time-step integration simulator and the one found in the event-driven simulator are perhaps the most significant difference between the two simulators. Instead of having the updateState and beginUpstroke operations, the time-step integration simulator uses the updateVoltage operation for all cells in all states during every time step. The getCurrent function is also now defined for every state (i.e., not only for the resting and stimulated states). The import/exportCell operations and the newton function have also been omitted. Other than these differences, the cell.c file is essentially the same in each of the two simulators.

4.2.3. main.c

The main.c file in the time-step integration simulator contains a much simpler version of the main simulation function found in the event-driven simulator. The function does not use a queue, does not have any mechanism for loading and saving states, and does not make use of any events.

The simulation loop runs on every time-step and carries out three operations during every iteration. First, it checks for the beginning or the end of external stimuli and applies the necessary changes to the affected cells. Second, it performs time-step integration on every cell by calling the updateVoltage operation. Finally, it checks to see if any form of output is required and performs it as necessary. The other aspects of the simulation process are effectively the same those found in the event-driven simulator.

5. References

[1] P. Ye and E. Entcheva and R. Grosu and S. A. Smolka, "Efficient Modeling of Excitable Cells Using Hybrid Automata". In *Proceedings of Computational Methods in System Biology*, 2005.