3 Real Time Architecture*

This chapter presents a brief summary of the TEXPLORE *algorithm before fully describing and presenting the real time RL architecture. First, I present a typical example of a sequential model-based RL architecture. Then I present details on using Monte Carlo Tree Search for planning, including a description of the modified version of the* UCT *algorithm (Kocsis and Szepesvári, 2006) that we use for planning. In Section 3.2, I present the parallel architecture for real time action, which puts model learning, planning, and acting on three parallel threads, such that actions can be taken as fast as required without being constrained by how long model updates or planning take. Finally, I summarize the chapter in Section 3.3.*

In this book, I introduce TEXPLORE, a sample-efficient model-based real time RL algorithm. When learning on robots, agents typically have very few samples to learn since the samples may be expensive, dangerous, or time-consuming. Therefore, learning algorithms for robots must be greedier than typical methods to exploit their knowledge in the limited time they are given. Since these algorithms must perform limited exploration, their exploration must be efficient and target state-actions that may be promising for the final policy. TEXPLORE achieves high sample efficiency by 1) utilizing the generalization properties of decision trees in building its model of the MDP, and 2) using random forests of those tree models to limit exploration to states that are promising for learning a good (but not necessarily optimal) policy quickly, instead of exploring more exhaustively to guarantee optimality. These two components constitute the key insights of the algorithm, and are explained in Chapter 4. Modifications to the basic decision tree model enable TEXPLORE to operate in domains with continuous state spaces as well as domains with action or observation delays.

The other key feature of the algorithm is that it can act in real time, at the frequencies required by robots (typically 5 - 20 Hz). For example, an RL agent controlling an autonomous vehicle must provide control signals to the gas and brake pedals immediately when a car in front of it slams on its brakes; it cannot stop to “think” about what to do. An alternative approach for acting in real time would be to learn off-line and then follow the learned policy in real time after the fact. However, it is desirable for the agent to be capable of learning on-line in-situ for the lifetime of the robot, adapting to new states and situations without pauses for computation. TEXPLORE combines a multi-threaded architecture with Monte

* This chapter contains material from two publications: (Hester et al., 2012; Hester and Stone, 2012b).

Algorithm 3.1. Sequential Model-Based Architecture

```
1: Input: $S, A$                                  ▷ $S$: state space, $A$: action space
2: Initialize $M$ to empty model
3: Initialize policy $\pi$ randomly
4: Initialize $s$ to a starting state in the MDP
5: loop
6:     Choose $a \leftarrow \pi(s)$
7:     Take action $a$, observe $r$, $s'$
8:     $M \Rightarrow$ UPDATE-MODEL($\langle s, a, s', r \rangle$)    ▷ Update model M with experience
9:     $\pi \leftarrow$ PLAN-POLICY($M$)              ▷ Exact planning on updated model
10:    $s \leftarrow s'$
11: end loop
```

Carlo Tree Search (MCTS) to provide actions in real time, by performing the model learning and planning in background threads while actions are returned in real time.

In this chapter, I introduce TEXPLORE's parallel architecture, enabling it to return actions in real time, addressing Challenge 4 of the *RL for Robotics Challenges*. Most current model-based RL methods use a sequential architecture such as the one shown in Figure 2.3 in Chapter 2. Pseudo-code for the sequential architecture is shown in Algorithm 3.1. In this sequential architecture, the agent receives a new state and reward; updates its model with the new transition $\langle s, a, s', r \rangle$ (i.e. by updating a tabular model or adding a new training example to a supervised learner); plans exactly on the updated model (i.e. by computing the optimal policy with a method such as value iteration (Sutton and Barto, 1998) or prioritized sweeping (Moore and Atkeson, 1993)); and returns an action from its policy. Since both the model learning and planning can take significant time, this algorithm is not real time. Alternatively, the agent may operate in batch mode (updating its model and planning on batches of experiences at a time), but this approach requires long pauses for the batch updates to be performed. Making the algorithm real time requires two modifications to the standard sequential architecture: 1) utilizing sample-based approximate planning (presented in Section 3.1) and 2) developing a novel parallel architecture (presented in Section 3.2). I later evaluate this planning method and parallel architecture in comparison with other approaches in Section 5.4.

3.1 Monte Carlo Tree Search (MCTS) Planning

The first component for providing actions in real time is to use an anytime algorithm for approximate planning, rather than performing exact planning using a method such as value iteration or prioritized sweeping. This section describes TEXPLORE's use of UCT for approximate planning as well as the modifications we have made to the algorithm. The standard UCT algorithm was presented in Section 2.2.4, but here we have modified UCT to use λ-returns, generalize values across depths in the search tree, maintain value functions between selected actions, and work in continuous domains. All of these changes are described in detail below.

TEXPLORE follows the approach of Silver et al. (2008) and Walsh et al. (2010) (among others) in using a sample-based planning algorithm from the MCTS family (such as Sparse Sampling (Kearns et al., 1999) or UCT (Kocsis and Szepesvári, 2006)) to plan *approximately*. These sample-based planners use a generative model to sample ahead from the agent's current state, updating the values of the sampled actions. These methods can be more efficient than dynamic programming approaches such as value iteration or policy iteration in large domains because they focus their updates on states the agent is likely to visit soon rather than iterating over the entire state space. While prioritized sweeping (Moore and Atkeson, 1993) improves upon the efficiency of value iteration by propagating value backups backwards through the state space, it still iterates over much of the state space rather than focusing computation on the states the agent is likely to visit soon.

The particular MCTS method that TEXPLORE uses is a variant of UCT (Kocsis and Szepesvári, 2006), which was presented in Algorithm 2.3 in Chapter 2. Our variation of UCT, called UCT(λ), is shown in Algorithm 3.2 and uses λ-returns, similar to the TD-SEARCH algorithm (Silver et al., 2012). UCT maintains visit counts for each state to calculate confidence bounds on the action-values. UCT differs from other MCTS methods by sampling actions more greedily by using the UCB1 algorithm (Auer et al., 2002), shown on Line 29. UCT selects the action with the highest upper confidence bound (with ties broken uniformly randomly). The upper confidence bound is calculated using the visit counts, c, to the state and each action, as well as the range of possible discounted returns in the domain, $\frac{r_{max}-r_{min}}{1-\gamma}$. Selecting actions this way drives the agent to concentrate its sampling on states with the best values, while still exploring enough to find the optimal policy.

UCT samples a possible trajectory from the agent's current state. On Line 30 of Algorithm 2.3, the model is queried for a prediction of the next state and reward given the state and selected action (QUERY-MODEL is described in detail later in Chapter 4 and shown in Algorithm 4.1). UCT continues sampling forward from the given next state. This process continues until the sampling has reached a terminal state or the maximum search depth, *maxDepth*. Then the algorithm updates the values of all the state-actions encountered along the trajectory. In normal UCT, the *return* of a sampled trajectory is the discounted sum of rewards received on that trajectory. The value of the initial state-action is updated towards this return, completing one *rollout*. The algorithm does many rollouts to obtain an accurate estimate of the values of the actions at the agent's current state. UCT is proven to converge to an optimal value function with respect to the model at a polynomial rate as the number of rollouts goes to infinity (Kocsis and Szepesvári, 2006).

We have modified UCT to update the state-actions using λ-returns, which average rewards received on the simulated trajectory with updates towards the estimated values of the states that the trajectory reached (Sutton and Barto, 1998). Informal experiments showed that using intermediate values of λ ($0 < \lambda < 1$) provided better results than using the default UCT without λ-returns.

In addition to using λ-returns, we have also modified UCT to generalize values across depths in the tree, since the value of a state-action in an infinite horizon discounted MDP is the same no matter when in the search it is encountered (due to the Markov property). One possible concern with this approach is that states at the bottom of the search tree may have poor value estimates because the search does not continue for many steps after reaching them. However, these states are not severely affected, since the λ-returns update them towards the values of the next states.

Most importantly, UCT is an anytime method, and will return better policies when given more time. By replacing the PLAN-POLICY call on Line 9 of Algorithm 3.1, which performs exact planning, with PLAN-POLICY from Algorithm 2.3, which performs approximate planning, the sequential architecture could be made faster. TEXPLORE's real time architecture, which is presented later in Algorithm 3.4, also uses UCT(λ) for planning.

UCT(λ) maintains visit counts for each state and state-action to determine confidence bounds on its action-values. When the model that UCT(λ) is planning on changes, its value function is likely to be incorrect for the updated model. Rather than re-planning entirely from scratch, the value function UCT(λ) has already learned can be used to speed up the learning of the value function for the new model. TEXPLORE's approach to re-using the previously learned value function is similar to the way Gelly and Silver (2007) incorporate off-line knowledge of the value function by providing an estimate of the value function and a visit count that represents the confidence in this value function. When UCT(λ)'s model is updated, the visit counts for all states are reset to a lower value that encourages UCT(λ) to explore again, but still enables UCT(λ) to take advantage of the value function learned for the previous model. The UCT-RESET procedure does so by resetting the visit counts for all state-actions to *resetCount*, which will be a small non-zero value. If the exact effect the change of the model would have on the value function is known, *resetCount* could be set based on this change, with higher values for smaller effects. However, TEXPLORE does not track the changes in the model, and even a small change in the model can have a drastic effect on the value function.

Some modifications must be made to use UCT(λ) on domains with continuous state spaces. One advantage of using UCT(λ) is that rather than planning ahead of time over a discretized state space, UCT(λ) can perform rollouts through the exact real-valued states the agent is visiting, and query the model for the real-valued state predictions. However, it cannot expect to ever visit the same real-valued state twice, nor can it maintain a table of values for an infinite number of states. Instead, it discretizes the state on Line 28 by discretizing each state feature into $nBins_i$ possible values. Since the algorithm is only using the discretization for the value function update, and not for the modeling or planning rollouts, it works well even on fine discretizations in high-dimensional domains.

Algorithm 3.2. PLAN: UCT(λ)

```
1: procedure UCT-INIT($S, A, maxDepth, resetCount, rmax, nBins, minVals, maxVals$)
2:     Initialize $Q(s,a)$ with zeros for all $s \in S, a \in A$
3:     Initialize $c(s,a)$ with ones for all $s \in S, a \in A$    ▷ To avoid divide-by-zero
4:     Initialize $c(s)$ with zeros for all $s \in S$    ▷ Visit Counts
5: end procedure

6: procedure PLAN-POLICY($M, s$)    ▷ Approx. planning from state $s$ using model $M$
7:     UCT-RESET()
8:     while time available do
9:         UCT-SEARCH($M, s, 0$)
10:    end while
11: end procedure

12: procedure UCT-RESET()    ▷ Lower confidence in v.f. since model changed
13:     for all $s_{disc} \in S_{disc}$ do    ▷ For all discretized states
14:         if $c(s_{disc}) > resetCount \cdot |A|$ then
15:             $c(s_{disc}) \leftarrow resetCount \cdot |A|$    ▷ $resetCount$ per action
16:         end if
17:         for all $a \in A$ do
18:             if $c(s_{disc}, a) > resetCount$ then
19:                 $c(s_{disc}, a) \leftarrow resetCount$
20:             end if
21:         end for
22:     end for
23: end procedure

24: procedure UCT-SEARCH($M, s, d$)    ▷ Rollout from state $s$ at depth $d$ using model $M$
25:     if TERMINAL or d = $maxDepth$ then
26:         return 0
27:     end if
28:     $s_{disc} \leftarrow$ DISCRETIZE($s, nBins, minVals, maxVals$)    ▷ Discretize state $s$
29:     $a \leftarrow \text{argmax}_{a'} \left( Q(s_{disc}, a') + 2 \cdot \frac{r_{max} - r_{min}}{1-\gamma} \cdot \sqrt{\frac{\log c(s_{disc})}{c(s_{disc}, a')}} \right)$    ▷ Ties broken randomly
30:     $(s', r) \leftarrow M \Rightarrow$QUERY-MODEL($s, a$)    ▷ Algorithm 4.1
31:     $sampleReturn \leftarrow r + \gamma$UCT-SEARCH($M, s', d+1$)    ▷ Continue rollout from state $s'$
32:     $c(s_{disc}) \leftarrow c(s_{disc}) + 1$    ▷ Update counts
33:     $c(s_{disc}, a) \leftarrow c(s_{disc}, a) + 1$
34:     $Q(s_{disc}, a') \leftarrow \alpha \cdot sampleReturn + (1-\alpha) \cdot Q(s_{disc}, a')$
35:     return $\lambda \cdot sampleReturn + (1-\lambda) \cdot \max_{a'} Q(s_{disc}, a')$    ▷ Use $\lambda$-returns
36: end procedure
```

Then the algorithm updates the value and visit counts for the discretized state on Lines 32 to 34.

3.1.1 Domains with Delay

We are particularly interested in applying TEXPLORE to robots and other physical devices, but one common problem with these devices is that their sensors and actuators have delays. For example, a robot's motors may be slow to start moving, and thus the robot may still be executing (or yet to execute) the last

Algorithm 3.3. UCT(λ) with delays

```
1: procedure SEARCH($M, s, history, d$)        ▷ Rollout from state $s$ with $history$
2:     if TERMINAL or d = $maxDepth$ then
3:         return 0
4:     end if
5:     $s_{disc} \leftarrow$ DISCRETIZE($s, nBins, minVals, maxVals$)
6:     $a \leftarrow \text{argmax}_{a'} \left( Q(s_{disc}, history, a') + 2 \cdot \frac{rmax}{1-\gamma} \cdot \sqrt{\frac{\log c(s_{disc}, history)}{c(s_{disc}, history, a')}} \right)$
7:     $augState \leftarrow \langle s, history \rangle$
8:     $(s', r) \leftarrow M \Rightarrow$QUERY-MODEL($augState, a$)
9:     PUSH($history, a$)        ▷ Keep last $k$ actions
10:    if LENGTH($history$) $> k$ then
11:        POP($history$)
12:    end if
13:    $sampleReturn \leftarrow r + \gamma$SEARCH($M, s', history, d+1$)
14:    $c(s_{disc}, history) \leftarrow c(s_{disc}, history) + 1$        ▷ Update counts
15:    $c(s_{disc}, history, a) \leftarrow c(s_{disc}, history, a) + 1$
16:    $Q(s_{disc}, history, a') \leftarrow \alpha \cdot sampleReturn + (1-\alpha) \cdot Q(s_{disc}, history, a')$
17:    return $\lambda \cdot sampleReturn + (1-\lambda) \cdot \max_{a'} Q(s_{disc}, history, a')$
18: end procedure
```

action given to it when the algorithm selects the next action. This delay is important, as the algorithm must take into account what the state of the robot will be when the action actually gets executed, rather than the state of the robot when the algorithm makes the action selection. TEXPLORE should model these delays and handle them efficiently.

Modeling and planning on domains with delay can be done by taking advantage of the k-Markov property (Katsikopoulos and Engelbrecht, 2003). While the next state and reward in these domains is not Markov with respect to the current state, it is Markov with respect to the previous k states. TEXPLORE takes advantage of the k-Markov property for planning by slightly modifying UCT(λ). Algorithm 3.3 shows the modified UCT(λ)-SEARCH algorithm. In addition to the agent's state, it also takes the history of k actions. While performing the rollout, it updates the history at each step (Lines 9 to 12), and uses the augmented state including history when querying the model (Line 8). States may have different optimal actions when reached with a different history, as different actions will be applied before the currently selected action takes place. This problem can be remedied by planning over an augmented state space that incorporates the k-action histories, shown in the visit count and value function updates in Lines 14 to 16. Katsikopoulos and Engelbrecht (2003) have shown that solving this augmented MDP provides the optimal solution to the delayed MDP. However, the state space increases by a factor of $|A|^k$. While this increase would greatly increase the computation required by a planning method such as value iteration that iterates over all the states, UCT(λ) focuses its updates on the states (or augmented state-histories) the agent is likely to visit soon, and thus its computation time is not greatly affected (demonstrated empirically in Section 5.3). Note that with $k = 0$, the *history* is $\emptyset$ and the action thread and UCT(λ) search methods presented here exactly match the ones presented in Algorithms 3.4 and 2.3, respectively. Later, in Section 5.3, we evaluate the performance of TEXPLORE's approach for handling delays in comparison with other approaches.

This version of UCT(λ) planning on the augmented state space is similar to the approach taken for planning inside the MC-AIXI algorithm (Veness et al., 2011). The difference is that their algorithm performs rollouts over a history of previous state-action-reward sequences, while TEXPLORE uses the current state along with only the previous k actions. One thing to note is that while TEXPLORE's approach is intended to address delays, it can also be used to address partial observability, if a sufficient k is chosen such that the domain is k-Markov.

Not only does this k-Markov approach to handling delay work well with UCT planning, it also works with our model learning approach. Later, in Section 4.1.2, we will describe how this approach applies to model learning.

3.2 Parallel Architecture

In addition to using MCTS for planning, we have developed a multi-threaded architecture, called the Real Time Model Based Architecture (RTMBA), for the agent to learn while acting in real time. Since UPDATE-MODEL and PLAN-POLICY can take significant computation (and thus also wall-clock time), they are placed in parallel threads in the background, as shown in Figure 3.1. A third thread selects actions as quickly as dictated by the robot control loop, while still being based on the most recent models and plans available. Pseudo-code for all three threads is shown in Algorithm 3.4. This architecture is general, allowing for any type of model learning method, and only requiring any method from the MCTS family for planning. In addition to enabling real time actions, this architecture enables the agent to take full advantage of multi-core processors by running each thread on a separate core. Similar approaches have been taken to parallelize MCTS planning and acting (Gelly et al., 2008; Chaslot et al., 2008; Méhat and Cazenave, 2011) by performing multiple rollouts in parallel, but they have not incorporated parallel model learning as well.

For the three threads to operate properly, they must share information while avoiding race conditions and data inconsistencies. The model learning thread must know which new transitions to add to its model, the planning thread must access the model being learned and know what state the agent is currently at, and the action thread must access the policy being planned. RTMBA uses mutex locks to control access to these variables, as summarized in Table 3.1.

The action thread (Lines 26 to 35) receives the agent's new state and reward, and adds the new transition experience, $\langle s, a, s', r\rangle$, to the *updateList* to be updated into the model. It then saves the agent's current state in *agentState* for use by the planner and returns the action determined by the agent's value function, Q. Since *updateList*, *agentState*, and Q are protected by mutex locks, it is possible that the action thread could have to wait for a mutex lock before it could proceed. However, *updateList* is only used by the model learning thread

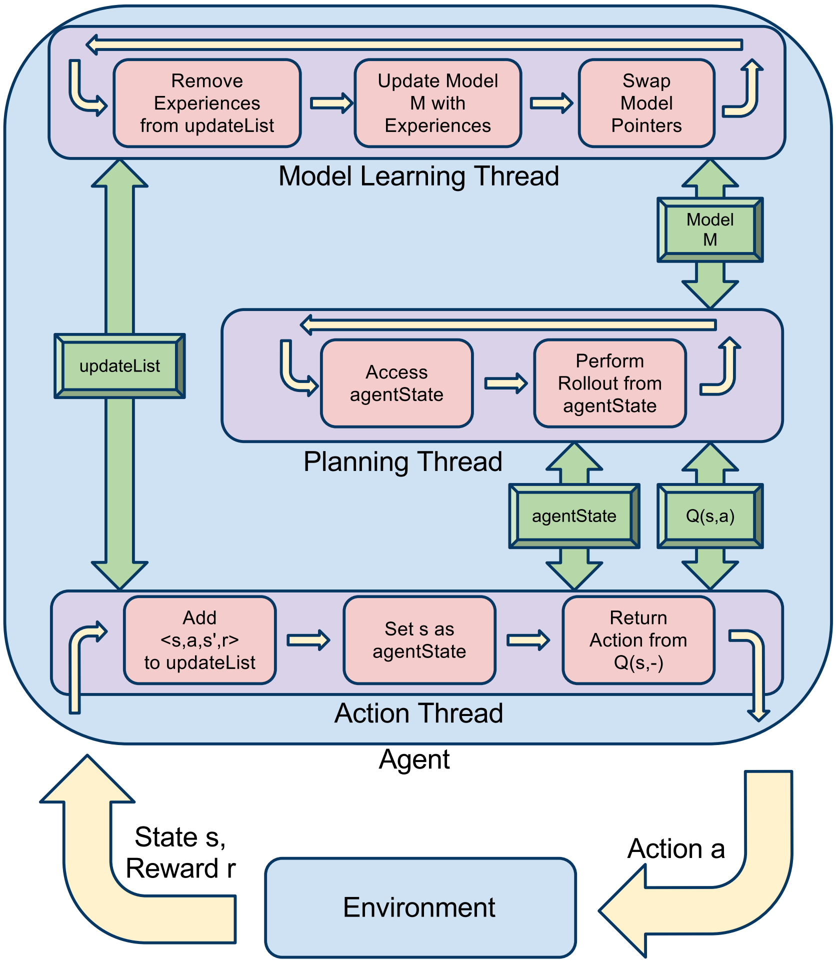

Fig. 3.1. A diagram of the parallel real time architecture for model-based RL

between model updates, *agentState* is only accessed by the planning thread between each rollout, and Q is under individual locks for each state. Thus, any given state is freely accessible most of the time. When the planner does happen to be using the same state the action thread wants, it releases it immediately after updating the values for that state. Therefore, there is never a long wait for mutex locks, and the action thread can return actions quickly when required.

The model learning thread (Lines 9 to 20) checks if there are any experiences in *updateList* to be added to its model. If there are, it makes a copy of its model to *tmpModel*, updates *tmpModel* with the new experiences, and clears *updateList*. Then it resets the planning visit counts to *resetCount* to lower the planner's confidence in the out-dated value function, which was calculated on an old model. Finally, on Line 18, it replaces the original model with the updated

Algorithm 3.4. Real Time Model-Based Architecture (RTMBA)

```
 1: procedure INIT                                              ▷ Initialize variables
 2:     Input: $S, A, nBins, minVals, maxVals$   ▷ $nBins$ is the # of discrete values
    for each feature
 3:     Initialize $s$ to a starting state in the MDP
 4:     $agentState \leftarrow s$
 5:     $updateList \leftarrow \emptyset$
 6:     Initialize $M$ to empty model
 7:     UCT-INIT()                                              ▷ Initialize Planner
 8: end procedure

 9: procedure MODELLEARNINGTHREAD                          ▷ Model Learning Thread
10:     loop                                     ▷ Loop, adding experiences to model
11:         while $updateList = \emptyset$ do
12:             Wait for experiences to be added to list
13:         end while
14:         $tmpModel \leftarrow M \Rightarrow$COPY          ▷ Make temporary copy of model
15:         $tmpModel \Rightarrow$UPDATE-MODEL($updateList$)   ▷ Update model $tmpModel$
    (Alg 4.1)
16:         $updateList \leftarrow \emptyset$                          ▷ Clear the update list
17:         UCT-RESET()                              ▷ Less confidence in current values
18:         $M \leftarrow tmpModel$                                ▷ Swap model pointers
19:     end loop
20: end procedure

21: procedure PLANNINGTHREAD                                     ▷ Planning Thread
22:     loop                                    ▷ Loop forever, performing rollouts
23:         UCT-SEARCH($M, agentState, 0$)                            ▷ Algorithm 2.3
24:     end loop
25: end procedure

26: procedure ACTIONTHREAD                                ▷ Action Selection Thread
27:     loop
28:         $s_{disc} \leftarrow$ DISCRETIZE($s, nBins, minVals, maxVals$)   ▷ Discretize state $s$
29:         Choose $a \leftarrow \text{argmax}_a Q(s_{disc}, a)$
30:         Take action $a$, Observe $r$, $s'$
31:         $updateList \leftarrow updateList \cup \langle s, a, s', r \rangle$   ▷ Add experience to update list
32:         $s \leftarrow s'$
33:         $agentState \leftarrow s$                  ▷ Set agent's state for planning rollouts
34:     end loop
35: end procedure
```

copy. The other threads can continue accessing the original model while the copy is being updated, since only the swapping of the models requires locking the model mutex. After updating the model, the model learning thread repeats, checking for new experiences to add to the model.

The model learning thread can call any type of model on Line 15, such as a tabular model (Brafman and Tennenholtz, 2001), a Gaussian Process regression

Table 3.1. This table shows all the variables that are protected under mutex locks in the real time architecture, along with their purpose and which threads use them

Variable	Threads	Use
updateList	Action, Model Learning	Store experiences to be updated into model
agentState	Action, Planning	Set current state to plan from
$Q(s, a)$	Action, Planning	Update policy used to select actions
M	Planning, Model Learning	Latest model to plan on

model (Deisenroth and Rasmussen, 2011), or the random forest model used by TEXPLORE, which is described in Chapter 4. Depending on how long the model update takes and how fast the agent is acting, the agent can add tens or hundreds of new experiences to its model at a time, or it can wait for long periods for a new experience. When adding many experiences at a time, full model updates are not performed between each individual action. In this case, the algorithm's sample efficiency is likely to suffer compared to that of sequential methods, but in exchange, it continues to act in real time.

Though TEXPLORE uses a variant of UCT, the planning thread can use any MCTS planning algorithm. The thread retrieves the agent's current state (*agentState*) and its planner performs a rollout from that state. The rollout queries the latest model, M, to update the agent's value function. The thread repeats, continually performing rollouts from the agent's current state. With more rollouts, the algorithm's estimates of action-values improve, resulting in more accurate policies. Even if very few rollouts are performed from the current state before the algorithm returns an action, many of the rollouts performed from the previous state should have gone through the current state (if the model is accurate), giving the algorithm a good estimate of the state's true action-values.

3.3 Chapter Summary

In this chapter, I have presented TEXPLORE's parallel real time architecture for model-based RL. This architecture parallelizes model learning, planning, and acting into three separate threads so that action selection can happen in real time, even if model learning or planning take more computation time. The architecture utilizes a sample-based anytime planning method, which improves as it is given time for more planning rollouts. In the next chapter, I will present the model learning method that is used within this architecture in the TEXPLORE algorithm.