BuildModel Algorithm

The following represents the algorithm to build a PRISM model using path commutation on a large number of input paths.

This algorithm is to be implemented in Java and relies heavily on the PRISM API.

PRISM is chosen as the probabilistic model checker for this project because of its extensive support for simulation.

Algorithm

Inputs:

- Newline-separated list of tab-separated reaction names representing reachable traces from the model's initial state to a target state
- PRISM model from which the traces were generated
- PRISM property to check states against

Outputs:

- State, transition, and label files in PRISM's desired format

Procedure:

1. Load model into PRISM engine
2. For each input path:
    1. Build the path:
        1. Set the PRISM engine to the initial state, plus any prefix reactions at deeper levels of recursion.
        2. Start saving a history of states along the path and an intersection of enabled reactions along the path.
        3. For each transition in the path:

        1. If the desired reaction has already been saved to the current state, follow the reaction to the next state and continue.
        2. Follow the reaction and check if the current state exists. If it does, save the reaction to it. If it does not, create a new state and save the reaction to it.
        3. Update the path history and intersection of enabled reactions.
    4. If by the end of the path, a target state is not reached, report an error.
2. Commute along the path:
    1. For each reaction r in the intersection of enabled reactions:
        1. Fire any prefix reactions from the initial state.
        2. Fire r to obtain the initial state of the novel path.
        3. If we are not at the maximum depth of recursion, start saving a history of states along the novel path and an intersection of enabled transitions along the novel path.
        4. For each state along the saved path:
            1. Fire the original path transition from the novel path to get state A.
            2. Fire r from the original path to get state B.
            3. If state A matches state B, save both reactions. Otherwise, save only the original path transition and not r.
            4. Check if state A exists. If it does, save the reaction to it. If it does not, create a new state and save the reaction to it.
            5. Update the path history and the intersection of enabled reactions.
        5. Recursively commute along each novel path until the maximum recursion depth is reached. Then continue.

Duplicate Checking

If we check duplicates against existing states, it is possible to search insane numbers of states. Given n variables with m_x possible values, it may be possible to search $m_1 \times m_2 \times m_3 \ldots \times m_n$ different states. So iterating through states is the most memory-efficient way to check duplicates, but it is too slow, especially as the state space expands.

If we use a modified search tree, the most comparisons we have to make is $m_1 + m_2 + m_3 \ldots + m_n$ but likely much less, since at each level, we compare every value at most once. This method also only adds $m_1 + m_2 + m_3 \ldots + m_n$ units to memory, and we never add memory we don't need to. This appears to be the most balanced option.

If we use a hash table, we use much more memory to store states in the hash table. We also use computational effort to calculate a hash for each state, and that number of operations may be comparable to the number of comparisons we'd have to make for a search tree. This is the most time-sensitive option, but it uses the most memory.

Desirable Future Features

- Check mean residence times and mean path times to get an estimate of probability
- Check mean times to see if model satisfies property within time bound
- Use PRISM's default way to store states (maybe -- they don't have all the features I want, like outgoing transition information, but I might be able to work around that)