1. Introduction

Typically manipulation of finite automata is performed using their state-transition-graph (STG) representations. Automata can also be represented using multi-valued networks. Given an automaton, whose next-state function and the set of accepting states are known, consider the network with a single MV latch shown in Figure 1. $F$ is the next-state logic that depends on the inputs and the current-state variable, and produces the next-state variable. It implements the next-state functions of the automaton. If the automaton is non-deterministic, $F$ is also non-deterministic. $\text{Acc}$ is a binary node, which only depends on the current state variable. This node produces value 1 if and only if the current state is accepting. Both $F$ and $\text{Acc}$ can be multi-level networks to compactly represent the automaton.

![Figure 1. Single-latch Network Representation for Finite Automaton](image)

More generally we can use MV network with multiple latches to represent an automaton. In such case the state encoding is not restricted to one-hot encoding and thus more compact representation may be obtained. However such representation does not fit for non-deterministic automaton. The main problem is SS( set simulation ) leads to loss of information and NS( normal simulation ) is needed to interpret the network. Consider the following example. The MV network contains 2 binary latches with inputs $\text{NS0}$ and $\text{NS1}$. Under some inputs it turns out that $\text{NS0}$ and $\text{NS1}$ have both 0 and 1 turned on. It is unknown if they corresponds to subset of states $\{00, 11\}$, or $\{01, 10\}$, or some 3-state subset like $\{11, 00, 10\}$. We need some pseudo input signals to control the combination of non-deterministic signals, which leads to the normal simulation. Unfortunately no many synthesis methods have been explored for NS. So we will only use such multiple-latch network to represent deterministic automata.

Since MV networks can represent automata compactly, it can be more efficient if we perform operations on finite automata on their MV network representations. Moreover, the optimization on automata mainly concerns about state minimization but the number of states is not good measurement for the complexity of the final circuit that implements the automaton. On the contrary, plenty of well developed synthesis algorithms on MV networks are available which can reduce the final circuit complexity efficiently.
The report is organized as following: Section 2 introduces algorithms developed on MV network for automata operations. Section 3 discusses implementation in MVSIS and future work.

2. Algorithms on MV Networks for Automata Operations

We consider the operations used in the main algorithm for this course, shown in Figure 2. $S$ and $F$ are given deterministic automata represented as STG. The network representation considered is multiple-latch network. As said in Section 1, this is the more general case compared with single-latch network.

1. $X := \text{complete}(S, \text{non-accepting})$
2. $X := \text{complement}(X)$
3. $X := \text{support}(X, (i,v,u,o))$
4. $X := \text{product}(F, X)$
5. $X := \text{support}(X, (v,u))$
6. $X := \text{determinize}(X)$
7. $X := \text{complement}(X)$
8. $X := \text{prefixClose} \& \text{uProgressive}(X, u)$

Figure 2. Algorithm for $X = (F \cdot S)_{u,v}$

2.1 STG to MV Network Conversion

The very first task is to convert STG to MV network. Currently the MV network studied must be well defined, which means for any primary input minterm, there is at least one corresponding output minterm. This actually implies that the automaton represented by the MV network must be complete. So operation $\text{complete}$ is included in the STG to network conversion.

1. $nStates := \text{ReadNumState}(sSTG)$
2. if( !IsComplete(sSTG)) $nStates++$
3. $nLatch := \lceil \log_2(nStates) \rceil$
4. $\text{stateToCodeTable} := \text{StateAssign}(sSTG, nLatch)$
5. $\text{LatchCreate}(sNet, nLatch)$
6. $\text{NextStateFunctions}(sNet, stateToCodeTable, sSTG)$
7. $\text{AccFunction}(sNet, stateToCodeTable, sSTG)$
8. $sNet->exdcNet := \text{unuseCodes}(stateToCodeTable)$

Figure 3. Algorithm for STG to Network Conversion

The outline algorithm is shown in Figure 3. Only binary latches are considered now. Line (6) generates the next state functions and initially they are single-leveled. All unused codes are summed up and saved in the exdc network, which can be used in later synthesis. This exdc network would need to be updated each time the network gets its latches changed.

2.2 Complement

Now we have a complete and deterministic automaton represented by a multiple-latch network. Operation $\text{complement}$ is simply to invert the $\text{Acc}$ function.
2.3 Product

As shown in Figure 4, the product of two automata represented by networks can be done trivially: AND the two Acc functions to produce the new Acc signal. The two exdc networks should be OR together to form the new exdc network. Such operation preserves the multi-level structure obtained from early synthesis.

![Figure 4](image)

**Figure 4. Product on Networks**

2.4 Hide & Determinize

As mentioned in Section 1, multiple-latch networks cannot represent non-deterministic automata conveniently. So we integrate hiding input variables and determinization based on subset construction into one step. This turns out to be the hardest step and several approaches are thought about.

2.4.1 Positional Encoding for Each State

Consider the original network contains N latches and we need to hide variable y. The most straightforward way is to introduce one latch for a state in the resulting network. Thus we have a unique code for each subset of states. Each latch has its input function to be the state transition function for its corresponding state, with y to be quantified, shown in Figure 5.

![Figure 5](image)

**Figure 5. Positional Encoding for Hide and Determinize**
The main advantage of this method is it does not require state space exploration. But the number of latches blows up after determinize. This method becomes infeasible even for networks with moderate sizes.

### 2.4.2 Partial Encoding

It has been noticed that actually for most automata the reachable subsets in determinize are only a small portion to all possible subsets. This motivates the partial encoding method for reducing the number of latches. Suppose the original automaton can be divided to several clusters, such that any reachable subset is contained in a cluster, shown in Figure 6. Then we can give each reachable subset a unique coding by: using a set of latches to encode which cluster this subset is in; the other set of latches to encode the subset in this cluster. The number of the latches for encoding clusters is determined by the number of clusters, and the number of the other set of latches is equal to the number of states in the largest cluster.

Note that the multi-level structure is only maintained for the logic feeding into latches that encode which subset inside a cluster, the “cluster choice” part logic is newly added. It is decided by the transitions between clusters.

![Figure 6. Partial Encoding](image)

Where or not this method can cut down the number of latches mainly depends on how well the automaton can be partitioned into clusters. An algorithm for fast computing clusters is shown in Figure 7. It begins with a single cluster that contains only the initial state. Then compute the reachable subsets from an existing cluster. If any subset gets interact with existing clusters, merge them into a new one. Keep doing this until no change happens.

Algorithm: StatePartition

Input: Automaton  
Output: List of clusters s.t. no reachable subset crosses clusters

begin

\[
\text{clusterList} := \{ \{ \text{init} \} \}; \text{change} := 1;
\]

while( change )

\{

\text{change} := 0; \text{set1 := Get( clusterList );}

\text{setList := nextState( set1 );}

\text{for each set2 in setList}

\{

\text{if( find( clusterList, set2)) continue;}

\text{intersectList := findIntersect( clusterList, set2 );}

\text{if( noEmpty( intersectList ))}

\{

\text{new := merge( intersectList, set2 );}

\text{listAdd( clusterList, new); change :=1;}

\}

\}
Figure 7. Fast Computation for Clusters

This algorithm is not implemented yet but is tried manually on two examples obtained from s298.aut (43 states) by hiding different set of variables. The results are shown in Figure 8. It turns out this algorithm is not stable. More examination on the first example shows that there is a state contained by most reachable subsets.

<table>
<thead>
<tr>
<th>Hidden Variables</th>
<th>Number of clusters</th>
<th>Number of states in largest cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>G1,G2,G17,G18,G19,G20,G21,G22,G23</td>
<td>11*</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 8. Cluster Results. * after merging

2.4.3 Further Discussion

In the above cluster computation algorithm we require clusters should not overlap. Actually this requirement is not necessary. If we can compute a set of overlapping covers s.t. any reachable subset is contained in a cover, then very likely less number of latches can be achieved. This part of algorithm is still under working.

2.4 prefixClose & uProgressive

To do prefixClose on an automaton means to remove non accepting states. However we cannot really remove some states from a network representation in order to keep the network well defined. The simplest way would be to have two ways to interpret the automaton represented by a network: 1. it represents the automaton containing all reachable states, including both accepting and non accepting states. 2. it represents the automaton only containing the accepting states, specified by the Acc node. A mark on the network can be introduced for this difference. Then the prefixClose operation reduces to changing the mark and adding the non accepting states to exdc. The latter step just means the transitions from non accepting states are no longer needed to be kept. The next states from non accepting states can be any states. This flexibility can be used to simplify the network.

For input progressive, it means to remove the accepting states that have no next state under some input pattern. On the network representation, this corresponds checking all states specified by the Acc function. Since the network is well defined, under any input pattern there will be a next state, but the states in exdc should be counted in. So to decide if an accepting state should be kept, we check its next state under all input patterns, if some falls into exdc, it should be removed from Acc and added to exdc. Keep such checking until the Acc function does not change.

So the basic idea is instead of really removing some states from the automata, we use the mark field to tell if they exist.

3. Implementations and Future Work

Currently only 3 commands are added to MVSIS, stg2net for STG to network conversion, net_complement for complementing automaton, net_product for computing the product of two automata represented by networks.

There are still a lot interesting topics to explore in the future. First of all is the overlapping cluster computation problem, which is critical for an efficient determinize algorithm. Second is the latch reduction problem that has not been mentioned above.
Functional dependency can be used here. Other problems such as state encoding in STG to network conversion also is worth studying.