CS 662
A General Data Structure for Efficient Minimization of Deterministic Finite Automata

Ondřej Lhoták
96040603

December 6, 2000

```
DFA-MINIMIZE($Q$ (set of states), $F$ (set of final states) )
1   equivalence class 1 $\leftarrow F$
2   equivalence class 2 $\leftarrow Q \setminus F$
3   while there is an equivalence class which must be split
4   do find an equivalence class which must be split
5      split it
```

Figure 1: Abstract DFA minimization algorithm

The Myhill-Nerode theorem states that for a deterministic finite automaton (DFA), there is a unique equivalent minimal DFA. Further, the states of this minimal DFA correspond to equivalence classes of the Myhill-Nerode equivalence relation. This suggests an obvious approach to minimizing DFA's by calculating this equivalence relation.

Many algorithms using this approach to minimize DFA's appear in the literature; Watson [4] gives a survey. Most of the well-known algorithms run in $\Theta(n^2)$ time or worse, where n is the number of states in the original DFA. The published algorithms running in $O(n \log n)$ time use complicated data structures, and are difficult to understand. It is the data structure, rather than the algorithm, which permits these approaches to run in $O(n \log n)$ time, yet the papers describe and explain the algorithms, with only very brief explanations of the data structures. The purpose of this paper is to identify the key features that make these algorithms efficient, and to present and explain a comprehensive data structure capturing all the information needed to minimize a DFA efficiently.

Watson's [4] survey paper classifies DFA minimization algorithms using the Myhill-Nerode relation according to whether they compute E, the Myhill-Nerode relation, D, its inverse, or $[Q]_E$, the set of equivalence classes of the relation. The algorithms computing D and E are simpler than ones computing $[Q]_E$, but they have no hope of running in better than $\Theta(n^2)$ time, since the size of the equivalence relation is quadratic in the number of states. Thus, an algorithm running in $O(n \log n)$ time must compute $[Q]_E$.

I am aware of three different published algorithms for minimizing DFA's in $O(n \log n)$ time. All three have the general structure shown in figure 1, but each fills in the details slightly differently.

The first is due to Hopcroft [3]. Hopcroft's paper gives his algorithm, a proof of correctness, and a proof of the running time including an informal description of the data structures used to implement the algorithm.

The second is due to Gries [2]. He claims that Hopcroft's algorithm and proof are "very difficult to understand." Gries presents pieces of his own algorithm along with simple lemmas proving the correctness of each piece. This makes the proof easier to understand. The algorithm of Gries is almost identical to that of Hopcroft (and he explains the small differences between the two); the main difference is in the presentation.

	Hopcroft	Gries	Blum
Partition	not used	not used	not used
Splittable Sets	L	L	K
Part	B	B	not used
State	s, t	s	not used
Transition Set	not named	not named	Δ'
Transition	not used	not used	L
State X Symbol	not used	not used	not named
StateSyms	not used	not used	Δ
StateSymsIn	not used	not used	Δ^{-1}
TransitionsOut	not used	not used	Γ
TransitionsIn	not used	not used	Γ^{-1}
TranSets	not used	not used	not mentioned

Table 1: Equivalent notation used for parts of the data structure

The third algorithm is due to Blum [1]. Its main distinguishing feature is the complexity of its data structure. In particular, Blum's data structure represents explicitly the transitions between equivalence classes of states, whereas those of Hopcroft and Gries only represent the equivalence classes themselves. Blum describes his algorithm only informally, in words, leaving it to the reader to fill in details. Blum's proof of correctness and running time is much simpler than those of Hopcroft and Gries, since many of the conditions that they had to prove are represented explicitly in Blum's data structure.

The data structure that I am presenting is a superset of the data structures of Hopcroft, Gries and Blum. Their data structures and algorithms were designed to be minimal, containing only the required information, to minimize the computational resources required. My data structure, on the other hand, represents as much information about the DFA as possible, while still being efficient enough to implement the minimization algorithms to run in $O(n \log n)$ time. Having all the information available makes the data structure easier to understand, and also makes it easier to use for implementing algorithms. Table 1 shows the correspondence between parts of my data structure and those of Hopcroft, Gries and Blum.

The data structure (see figure 2) consists of sets of objects of different types, most of them connected with one-to-many relations. Hopcroft, Gries, and Blum represent each of these relations using different structures. For example, some may be implemented as simple linked lists, others need to be doubly-linked lists, still others need to also keep track of their size, or allow access to the containing object from each of the contained objects. To simplify the structure, I implemented all the relations using the same structure: a circular doubly-linked list with a dummy node, keeping track of its size, and maintaining a pointer to the object owning the list. This supports all the operations needed by the algorithms. The same list code can be used for all the relations, rather than implementing a different type of list for each relation.

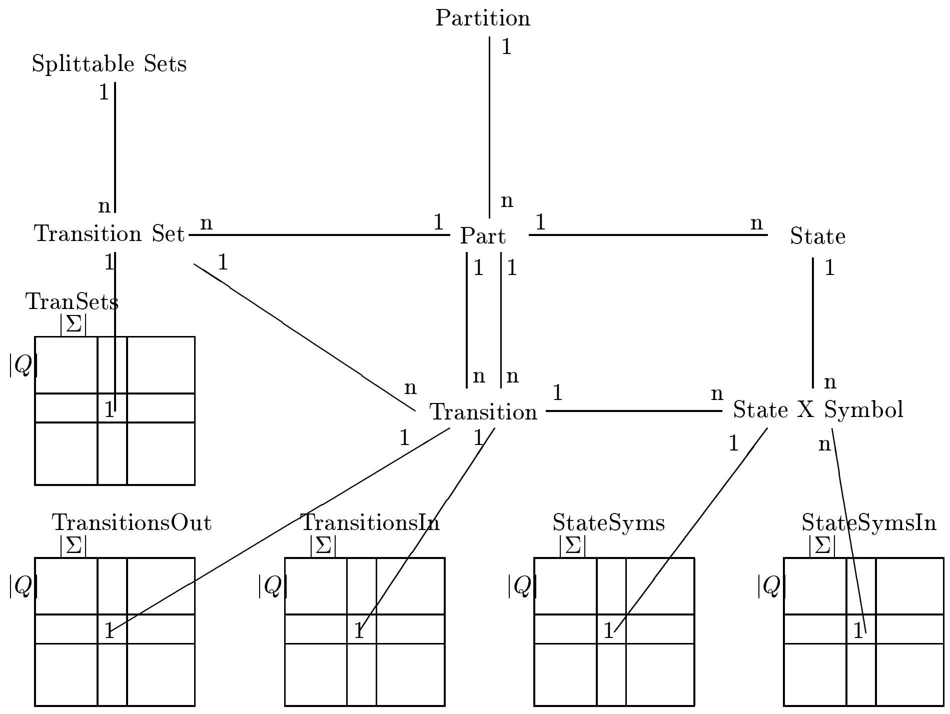

Figure 2: A general data structure for DFA minimization

The main object in the data structure is a Partition. This represents a partitioning of the states of the DFA into a set of equivalence classes, represented by Part objects. Each equivalence class contains the set of State objects forming the equivalence class. It also contains a set of Transition Set objects. Each of these objects represents a set of transitions leading *out of* states in the equivalence class on a given symbol. Thus, a Part object contains one Transition Set for each symbol in the alphabet. Each Transition Set contains a set of Transition objects. A Transition object represents a transition on a certain symbol from one equivalence class into another single equivalence class. Note that an equivalence class needs to be split if and only if there is a symbol for which there are transitions into more than one equivalence class. Put another way, a Part needs to be split if and only if there are at least two Transition objects in any of its Transition Set objects. The Splittable Sets object contains a list of all Transition Set objects containing at least two Transitions, facilitating the selection of a Transition Set which can be used to split its Part. Each Transition object contains a list of the states in its source equivalence class with transitions into the destination equivalence class on the given symbol. These are exactly the states that need to be moved into a new equivalence class when the source equivalence class is split. Because each state can appear in different Transition objects, one for each symbol, I represent the states with a special State X Symbol object.

Finally, the data structure contains five arrays. The StateSyms array is simply a mapping from states and symbols to the corresponding State X Symbol object. The StateSymsIn array contains lists of State X Symbol objects from which there are transitions on a given symbol into a given state. The TransitionsOut and TransitionsIn arrays are used to find the Transition object representing the transitions from a class Q_i into a class Q_j on symbol a. Representing this directly would require an array of size n^2k, where n is the number of states (the maximum number of equivalence classes) and k is the number of symbols. Such an array would take more than $O(n \log n)$ time to initialize and manipulate. Therefore, two arrays are used: TransitionsIn and TransitionsOut. TransitionsOut maps (Q_i, a) to the last created Transition from Q_i on symbol a. Fortunately, the algorithm only ever needs to find Transitions into the last equivalence class created. When using the array TransitionsOut, the algorithm must always check whether the Transition it finds in fact goes into the desired equivalence class, since if there is no Transition into the most recently created equivalence class, the array will point to an old Transition. Similarly, the TransitionsIn array maps (a, Q_j) to the last created Transition into Q_j on symbol a, and the algorithm must check whether this Transition actually originates from the desired equivalence class Q_i. Finally, the TranSets array allows us to find the TransitionSet object given the Part from which the transition originates, and the symbol on which the transition is taken.

Figures 3 through 9 give a formal description of an $O(n \log n)$ algorithm for minimizing DFA's using the data structure just described. The algorithm is based on the one due to Blum. Enough detail is given so that each line could be translated to one line of C++ or other programming language.

```
DFA-MINIMIZE($Q$ (set of states), $F$ (set of final states), $\delta$ (transition function) )
    // Initialize the data structure
  INITIALIZE($Q, \delta$)
    // Create a new part for final states
  $newpart \leftarrow$ CREATE-PART()
  for $state \in F$
  do MOVE-STATE($state, newpart$)
    // Minimize the DFA
  while $SplittableSets.splittablesets\_transitionset.size() > 0$
  do $transet \leftarrow SplittableSets.splittablesets\_transitionset.first()$
      $tran1 \leftarrow transet.transitionset\_transition.first()$
      $tran2 \leftarrow transet.transitionset\_transition.first().next()$
      if $tran1.transition\_statesym.size() > tran2.transition\_statesym.size()$
        then SWAP($tran1, tran2$)
      $newpart \leftarrow$ CREATE-PART()
      for $statesym \in tran1.transition\_statesym$
      do $state \leftarrow statesym.state\_statesym.getOwner()$
          MOVE-STATE($state, newpart$)
    // Part objects are now states of minimized DFA
    // Transition objects are now transitions of minimized DFA
```

Figure 3: DFA-Minimize

The algorithm begins by initializing the data structure with a single equivalence class containing all the states. It then uses MOVE-STATE to move all final states into a second equivalence class. Then, as long as there are equivalence classes which need to be split, the algorithm splits them, always moving the smaller half of the states in an equivalence class into a new equivalence class.

The MOVE-STATE procedure is the most complicated part of the algorithm, as it updates all parts of the data structure to reflect moving a state into a different equivalence class. First, it moves the actual State object into the new Part object. It must then update information for all transitions into and out of the moved state. This is accomplished by the two main loops in MOVE-STATE.

Each line in the algorithm (other than loops) takes a constant amount of running time. It is easy to verify that ADD-TRANSITION-TO-TRANSITIONSET, REMOVE-TRANSITION-FROM-TRANSITIONSET, and REMOVE-STATESYM-FROM-TRANSITION run in constant time. MOVE-STATE and CREATE-PART run in $O(k)$ time, where k is the number of symbols in the alphabet Σ. INITIALIZE runs in $O(kn)$ time, where n is the number of states in Q.

Finally, I am left to analyze the running time of DFA-MINIMIZE. This running time is bounded by the total time spent executing MOVE-STATE. Whenever any state is moved into a different part, the new part contains at most half as many states as the part that the state was moved from. Therefore, each state can be moved at most $\log_2 n$ times. Since there are n states, MOVE-STATE is

```
MOVE-STATE($state, newpart$)
  // move the actual state object
  $oldpart \leftarrow state.part\_state.getOwner()$
  $state.part\_state.remove(state)$
  $newpart.part\_state.add(state)$
  // now update transitions out of the moved state
  for $statesym \in state.state\_statesym$
  do $oldtransition \leftarrow statesym.transition\_statesym.getOwner()$
     $oldtopart \leftarrow oldtransition.topart\_transition.getOwner()$
     REMOVE-STATESYM-FROM-TRANSITION($statesym$)
     $newtransition \leftarrow TransitionsOut(oldtopart, statesym.symbol)$
     if $newtransition.frompart\_transition.getOwner() \neq newpart$
       then $oldtopart \leftarrow newtransition.topart\_transition.getOwner()$
            $newtransition \leftarrow$ new $Transition$
            $newpart.frompart\_transition.add(newtransition)$
            $oldtopart.topart\_transition.add(newtransition)$
            $transet \leftarrow TranSets(newpart, statesym.symbol)$
            ADD-TRANSITION-TO-TRANSITIONSET($newtransition, transet$)
            $TransitionsOut(oldtopart, statesym.symbol) \leftarrow newtransition$
     $newtransition.transition\_statesym.add(statesym)$
  // now update transitions into the moved state
  for $symbol \in \Sigma$
  do for $statesym \in StateSymsIn(state, symbol)$
     do $oldtransition = statesym.transition\_statesym.getOwner()$
        $oldfrompart = oldtransition.frompart\_transition.getOwner()$
        REMOVE-STATESYM-FROM-TRANSITION($statesym$)
        $newtransition \leftarrow TransitionsIn(oldfrompart, statesym.symbol)$
        if $newtransition.topart\_transition.getOwner() \neq newpart$
          then $oldfrompart \leftarrow newtransition.frompart\_transition.getOwner()$
               $newtransition \leftarrow$ new $Transition$
               $newpart.topart\_transition.add(newtransition)$
               $oldfrompart.frompart\_transition.add(newtransition)$
               $transet \leftarrow TranSets(oldfrompart, statesym.symbol)$
               ADD-TRANSITION-TO-TRANSITIONSET($newtransition, transet$)
               $TransitionsIn(oldfrompart, statesym.symbol) \leftarrow newtransition$
        $newtransition.transition\_statesym.add(statesym)$
```

Figure 4: Move-State

```
INITIALIZE($Q, \delta$)
 1  $Partition \leftarrow$ new $Partition$
 2  $SplittableSets \leftarrow$ new $SplittableSets$
 3  $part \leftarrow$ CREATE-PART()
 4  for $q \in Q$
 5  do $state \leftarrow$ new $State$
 6      $part.part\_state.add(state)$
 7      for $symbol \in \Sigma$
 8      do $statesym \leftarrow$ new $StateSym$
 9          $state.state\_statesym.add(statesym)$
10          $StateSyms(state, symbol) \leftarrow statesym$
11  for $state \in Q$
12  do for $symbol \in \Sigma$
13      do $StateSymsIn(\delta(state, symbol), symbol).add(StateSyms(state, symbol))$
14  for $symbol \in \Sigma$
15  do $transition \leftarrow$ new $Transition$
16      $part.frompart\_transition.add(transition)$
17      $part.topart\_transition.add(transition)$
18      $TransitionsIn(part, symbol) \leftarrow transition$
19      $TransitionsOut(part, symbol) \leftarrow transition$
20      ADD-TRANSITION-TO-TRANSITIONSET($transition, TranSets(part, symbol)$)
21      for $state \in Q$
22      do $transition.transition\_statesym.add(StateSyms(state, symbol))$
```

Figure 5: Initialize

```
CREATE-PART()
1  $part \leftarrow$ new $Part$
2  $Partition.partition\_part.add(part)$
3  for $symbol \in \Sigma$
4  do $transet \leftarrow$ new $TransitionSet$
5      $part.part\_transitionset.add(transet)$
6      $TranSets(part, symbol) \leftarrow transet$
```

Figure 6: Create-Part

```
ADD-TRANSITION-TO-TRANSITIONSET($transition, transet$)
1  $transet.transitionset\_transition.add(transition)$
2  if $transet.transitionset\_transition.size() = 2$
3     then $SplittableSets.splittablesets\_transitionset.add(transet)$
```

Figure 7: Add-Transition-To-TransitionSet

```
REMOVE-TRANSITION-FROM-TRANSITIONSET($transition$)
1  $transet \leftarrow transition.transitionset\_transition.getOwner()$
2  $transition.transitionset\_transition.remove(transition)$
3  if $transet.transitionset\_transition.size() = 1$
4     then $SplittableSets.splittablesets\_transitionset.remove(transet)$
```

Figure 8: Remove-Transition-From-TransitionSet

```
REMOVE-STATESYM-FROM-TRANSITION($statesym$)
1  $transition \leftarrow statesym.transition\_statesym.getOwner()$
2  $statesym.transition\_statesym.remove(statesym)$
3  if $transition.transition\_statesym.size() = 0$
4     then REMOVE-TRANSITION-FROM-TRANSITIONSET($transition$)
5
```

Figure 9: Remove-StateSym-From-Transition

called $O(n \log n)$ times. Each call to MOVE-STATE takes $O(k)$ time, so the total running time if DFA-MINIMIZE is $O(kn \log n)$. For more details, see [1].

To show correctness, we first note that the SplittableSets list contains all TransitionSet objects containing two or more Transition objects. To verify this, notice that Transition objects are only added to or removed from TransitionSet objects in the functions ADD-TRANSITION-TO-TRANSITIONSET and REMOVE-TRANSITION-FROM-TRANSITIONSET, and that these functions correctly update SplittableSets. I argued in the description of the data structure that there are equivalence classes to split if and only if there are at least two Transition objects in any TransitionSet object, that is, if SplittableSets is not empty. Since SplittableSets must be empty for the algorithm to terminate, it can only terminate once the correct answer is found. Finally, note that only equivalence classes which have to be split (those in SplittableSets) are ever split, and that creating a new Part and moving the States into it using MOVE-STATE correctly splits an equivalence class. Again, for more details, see [1].

I have presented and explained a general data structure which can be used to implement DFA minimization algorithms running in $O(n \log n)$ time, where n is the number of states in the DFA. This structure could also be used for other efficient DFA manipulation algorithms. As an example, I have formalized and modified Blum's DFA minimization algorithm to use my general data structure, and implemented it in C++. The C++ code is given in the appendices.

References

[1] Norbert Blum. An $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*,

57(2):65–69, January 1996.

[2] David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[3] Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations, Ed. by Zvi Kohavi and Azaria Paz, Academic Press.* 1971.

[4] B. W. Watson. A taxonomy of finite automata minimization algorithms. Report, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1994.

A Code for one-to-many relations

```
/* implementation of a doubly-linked list template class */
/* each function takes O(1) time */

template <class Owner, class Elem> class Reln
{
    public:
        // Member access functions
            Elem* getElem() {
            return _elem;
        }

        Owner* getOwner() {
            return (Owner*) _root->_elem;
        }

        Reln<Owner, Elem>* next() {
            return _next;
        }

        Reln<Owner, Elem>* prev() {
            return _prev;
        }

        Reln<Owner, Elem>* first() {
            return _root->next();
        }

        int size() {
            return _root->_size;
        }

        // is_root_node - returns true iff this node
        // is the root (dummy) node
        bool is_root_node() {
            return _root == this;
        }

        // insert - inserts a new node into the list after the current node
        void add( Reln<Owner, Elem>* new_node ) {
            new_node->unlink(); // make sure it's not linked

            new_node->_next = _root->_next; // make new node point into the list
            new_node->_prev = _root;
            new_node->_root = _root;

            _root->_next->_prev = new_node;
                // make the list point to the new node
            _root->_next = new_node;

            _root->_size++;
        }

        // unlink - removes this node from any list in which it may be
        virtual void unlink() {
            _root->_size--;
            _next->_prev = _prev; // remove me from the list
            _prev->_next = _next;

            _next = this; // make me no longer point to the list
            _prev = this;
            _root = this;
            _root->_size = 1;
        }

        // constructors
        Reln<Owner, Elem>( Owner* owner ) {
            _elem = (Elem*) owner;
            _next = this;
            _prev = this;
            _root = this;
            _size = 0;
        }

        Reln<Owner, Elem>( Elem* elem ) {
            _elem = elem;
            _next = this;
            _prev = this;
            _root = this;
            _size = 1;
        }

    private:
        Elem* _elem;
        Reln<Owner, Elem>* _next;
        Reln<Owner, Elem>* _prev;
        Reln<Owner, Elem>* _root;
        int _size;
};

// macro to iterate over all elements in the list
#define FORALL( owner_type, elem_type, elem, list ) \
    for( Reln<owner_type,elem_type>* iterator = (list).first(); \
         !iterator->is_root_node(); ) { \
        elem_type* elem = iterator->getElem(); \
iterator = iterator->next();

#define ADD( relation, element ) \
    relation.add( &(element->relation) )
```

B Declarations of data structure

```
struct Partition;
struct Part;
struct TransitionSet;
struct Transition;
struct StateSym;
struct State;
struct SplittableSets;

struct Partition {
    Reln<Partition, Part> partition_part;

    Partition()
        : partition_part( this )
    {}
};

struct Part {
    Reln<Part, Transition> frompart_transition;
    Reln<Part, Transition> topart_transition;
    Reln<Part, TransitionSet> part_transitionset;
    Reln<Part, State> part_state;
    Reln<Partition, Part> partition_part;

    int part;

    Part( int part )
        : frompart_transition( this )
        , topart_transition( this )
        , part_transitionset( this )
        , part_state( this )
        , partition_part( this )
        , part( part )
    {}
};

struct TransitionSet {
    Reln<TransitionSet, Transition> transitionset_transition;
    Reln<Part, TransitionSet> part_transitionset;
    Reln<SplittableSets, TransitionSet> splittablesets_transitionset;

    TransitionSet()
        : transitionset_transition( this )
        , part_transitionset( this )
        , splittablesets_transitionset( this )
    {}
};

struct Transition {
    Reln<TransitionSet, Transition> transitionset_transition;
    Reln<Part, Transition> frompart_transition;
    Reln<Part, Transition> topart_transition;
    Reln<Transition, StateSym> transition_statesym;

    Transition()
        : transitionset_transition( this )
        , frompart_transition( this )
        , topart_transition( this )
        , transition_statesym( this )
    {}
};

struct StateSym {
    Reln<Transition, StateSym> transition_statesym;
    Reln<State, StateSym> state_statesym;
    Reln<void, StateSym> statesymsin_statesym;
    int state;
    int symbol;

    StateSym( int state, int symbol )
        : transition_statesym( this )
        , state_statesym( this )
        , statesymsin_statesym( this )
        , state( state )
        , symbol( symbol )
    {}
};

struct State {
    Reln<State, StateSym> state_statesym;
    Reln<Part, State> part_state;

    int state;

    State( int state )
        : state_statesym( this )
        , part_state( this )
        , state( state )
    {}
};

struct SplittableSets {
    Reln<SplittableSets, TransitionSet> splittablesets_transitionset;

    SplittableSets()
        : splittablesets_transitionset( this )
    {}
};

Transition* TransitionsOut[ NUM_STATES ][ NUM_SYMBOLS ];
```

```
Transition* TransitionsIn[ NUM_STATES ][ NUM_SYMBOLS ];
StateSym* StateSyms[ NUM_STATES ][ NUM_SYMBOLS ];
Reln<void,StateSym>* StateSymsIn[ NUM_STATES ][ NUM_SYMBOLS ];
TransitionSet* TranSets[ NUM_STATES ][ NUM_SYMBOLS ];
```

C Code for DFA minimization algorithm

```
static Partition* partition;
static SplittableSets* splittablesets;
static int last_part = -1;
static State* states[NUM_STATES];

void DFAMinimize( int NUM_FINAL_STATES, int delta[NUM_STATES][NUM_SYMBOLS] );
void MoveState( State* state, Part* newpart );
void Initialize( int delta[NUM_STATES][NUM_SYMBOLS] );
Part* CreatePart();
void AddTransitionToTransitionSet( Transition* transition,
                                   TransitionSet* transet );
void RemoveTransitionFromTransitionSet( Transition* transition );
void RemoveStateSymFromTransition( StateSym* statesym );

void DFAMinimize( int NUM_FINAL_STATES, int delta[NUM_STATES][NUM_SYMBOLS] ) {
    // assume that states 0..NUM_FINAL_STATES are final states

    // Initialize the data structure
    Initialize( delta );

    // Create a new part for final states
    Part* newpart = CreatePart();
    for( int state = 0; state < NUM_FINAL_STATES; state++ ) {
        MoveState( states[state], newpart );
    }

    // Minimize the DFA
    while( splittablesets->splittablesets_transitionset.size() > 0 ) {
        TransitionSet* transet =
            splittablesets->splittablesets_transitionset.first()->getElem();
        Transition* tran1 =
            transet->transitionset_transition.first()->getElem();
        Transition* tran2 =
            transet->transitionset_transition.first()->next()->getElem();

        if( tran1->transition_statesym.size() >
            tran2->transition_statesym.size() ) {
            Transition* temp = tran1; tran1 = tran2; tran2 = temp;
        }

        Part* newpart = CreatePart();
        FORALL( Transition, StateSym, statesym, tran1->transition_statesym )

            State* state = statesym->state_statesym.getOwner();
            MoveState( state, newpart );
        }
    }
}

void MoveState( State* state, Part* newpart ) {
    // move the actual state object
    Part* oldpart = state->part_state.getOwner();
    state->part_state.unlink();
    newpart->ADD( part_state, state );

    // now update transitions out of the moved state
    FORALL( State, StateSym, statesym, state->state_statesym )
        Transition* oldtransition = statesym->transition_statesym.getOwner();
        Part* oldtopart = oldtransition->topart_transition.getOwner();
        RemoveStateSymFromTransition( statesym );
        Transition* newtransition =
                TransitionsOut[ oldtopart->part ][ statesym->symbol ];
        if( !newtransition || newtransition->frompart_transition.getOwner() != newpart ) {
            newtransition = new Transition;
            newpart->ADD( frompart_transition, newtransition );
            oldtopart->ADD( topart_transition, newtransition );
            TransitionSet* transet =
                TranSets[ newpart->part ][ statesym->symbol ];
            AddTransitionToTransitionSet( newtransition, transet );
            TransitionsOut[ oldtopart->part ][ statesym->symbol ] =
                newtransition;
        }
        newtransition->ADD( transition_statesym, statesym );
    }

    // now update transitions into the moved state
    for( int symbol = 0; symbol < NUM_SYMBOLS; symbol++ ) {
        FORALL( void, StateSym, statesym,
            *(StateSymsIn[ state->state ][ symbol ]) )
            Transition* oldtransition = statesym->transition_statesym.getOwner();
            Part* oldfrompart = oldtransition->frompart_transition.getOwner();
            RemoveStateSymFromTransition( statesym );
            Transition* newtransition =
                TransitionsIn[ oldfrompart->part ][ statesym->symbol ];
            if( !newtransition || newtransition->topart_transition.getOwner() != newpart ) {
                newtransition = new Transition;
                newpart->ADD( topart_transition, newtransition );
                oldfrompart->ADD( frompart_transition, newtransition );
                TransitionSet* transet =
                    TranSets[ oldfrompart->part ][ statesym->symbol ];
                AddTransitionToTransitionSet( newtransition, transet );
                TransitionsIn[ oldfrompart->part ][ statesym->symbol  ] =
                    newtransition;
            }
            newtransition->ADD( transition_statesym, statesym );
        }
    }
}

void Initialize( int delta[NUM_STATES][NUM_SYMBOLS] ) {
    partition = new Partition;
    splittablesets = new SplittableSets;
    Part* part = CreatePart();
    for( int q = 0; q < NUM_STATES; q++ ) {
        for( int symbol = 0; symbol < NUM_SYMBOLS; symbol++ ) {
            StateSymsIn[q][symbol] = new Reln<void, StateSym>( (void*) NULL );
        }
    }
    for( int q = 0; q < NUM_STATES; q++ ) {
        State* state = new State( q );
        states[q] = state;
        part->ADD( part_state, state );
        for( int symbol = 0; symbol < NUM_SYMBOLS; symbol++ ) {
            StateSym* statesym = new StateSym( q, symbol );
            state->ADD( state_statesym, statesym );
            StateSyms[q][symbol] = statesym;
        }
    }
    for( int q = 0; q < NUM_STATES; q++ ) {
        for( int symbol = 0; symbol < NUM_SYMBOLS; symbol++ ) {
            StateSymsIn[ delta[q][symbol] ][symbol]->
                add( &(StateSyms[q][symbol]->statesymsin_statesym) );
        }
    }
    for( int symbol = 0; symbol < NUM_SYMBOLS; symbol++ ) {
        Transition* transition = new Transition;
        part->ADD( frompart_transition, transition );
        part->ADD( topart_transition, transition );
        TransitionsIn[last_part][symbol] = transition;
        TransitionsOut[last_part][symbol] = transition;
        AddTransitionToTransitionSet( transition,
            TranSets[last_part][symbol] );
        for( int q = 0; q < NUM_STATES; q++ ) {
            transition->ADD( transition_statesym, StateSyms[q][symbol] );
        }
    }
}

Part* CreatePart() {
    Part* part = new Part( ++last_part );
    partition->ADD( partition_part, part );
    for( int symbol = 0; symbol < NUM_SYMBOLS; symbol++ ) {
        TransitionSet* transet = new TransitionSet;
        part->ADD( part_transitionset, transet );
        TranSets[ last_part][ symbol ] = transet;
    }
    return part;
}

void AddTransitionToTransitionSet( Transition* transition,
                                   TransitionSet* transet ) {

    transet->ADD( transitionset_transition, transition );
    if( transet->transitionset_transition.size() == 2 ) {
        splittablesets->ADD( splittablesets_transitionset, transet );
    }
}

void RemoveTransitionFromTransitionSet( Transition* transition ) {
    TransitionSet* transet =
        transition->transitionset_transition.getOwner();
    transition->transitionset_transition.unlink();
    if( transet->transitionset_transition.size() == 1 ) {
        transet->splittablesets_transitionset.unlink();
    }
}

void RemoveStateSymFromTransition( StateSym* statesym ) {
    Transition* transition = statesym->transition_statesym.getOwner();
    statesym->transition_statesym.unlink();
    if( transition->transition_statesym.size() == 0 ) {
        RemoveTransitionFromTransitionSet( transition );
    }
}
```