Chapter 3. Implementation of Branching Bisimulation Minimization

In this chapter we will first introduce the abstract procedures of Branching Bisimulation Minimization based on the algorithm proposed by Bouali [27]. Then we present the data structure inspired from [26][27]. The remaining sections elaborate the detailed implementation of each procedure.

3.1 Abstract Procedures

Preprocess: Before doing branching bisimulation minimization, we have to merge those states formed a τ-cycle into one state, but preserve necessary transitions. The algorithm of finding *strong connected components* in a graph [31][32] is used to solve this problem.

Main Loop: This is the main procedure for branching bisimulation minimization, which is revised from the general algorithm of RCP problem (see section 2.5). It repeats the work of finding unstable blocks and refining them until the whole partition is stable. It calls the procedure of *checking unstable blocks* to accomplish its work.

Check Unstable Blocks: In this procedure, it takes a block *B*' as a *splitter* to check whether other blocks having transitions ending in *B*' are stable. To do this work,

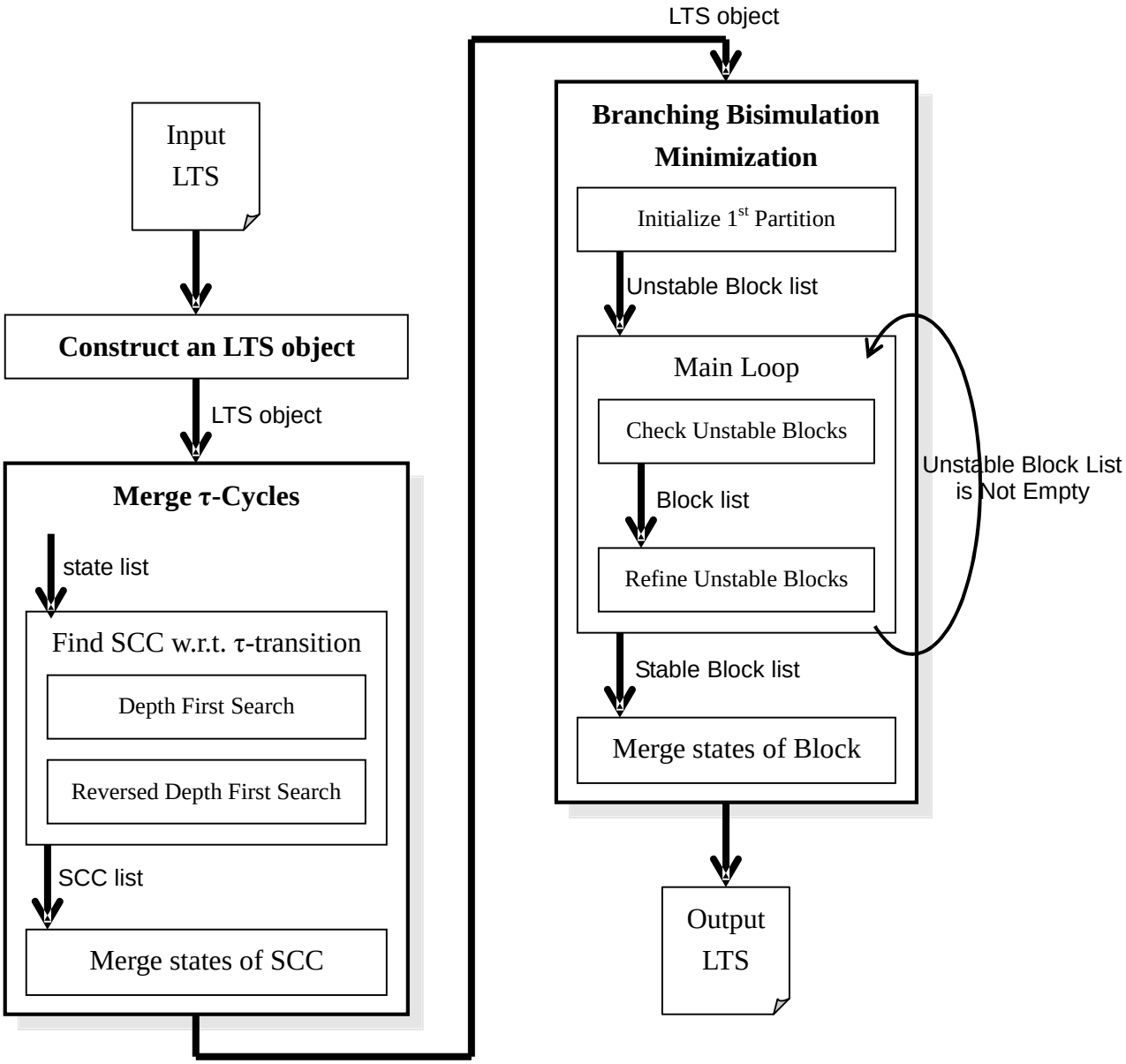


Figure 3.1. Design Framework of Branching Bisimulation Minimization.

it invokes the procedures of *obtaining unstable blocks by instability notion* and *refining unstable blocks.*

Obtain Unstable Blocks by Instability Notion: The main work of this procedure is to collect those *potentially* unstable blocks, which have outgoing transitions reaching to states in *B*' via a particular action, into a block list for the next procedure of *refining*

unstable blocks.

Refine Unstable Blocks: In this procedure, it tries to refine every potentially unstable block in the block list gained by the previous procedure. If the block is instable corresponding to the instability notion, then the block will be split into two minor blocks.

3.2 Data Structure

We adopt the data type introduced by Bouali in [27] for our implementation in C++ with STL (Standard Template Library). Bouali represented an LTS implicitly by a structured partition containing all its states and transitions. Below we describe the idea of the design inspired from his notion.

A partition is a list of blocks, and **a block** *B* can be seen as a structure containing

[Data Members]
- `BottomList:` the state list of ***bottom***(*B*).
- `NonBottomList:` the state list of { *B*-***bottom***(*B*) }.
- `TransitionList:` the list of transitions having a target state ending in *B*.

[Member Functions]
- `resetMark:` reset all states' `mark` to false.
- `resetBlockPtr:` correct all states' `BlockPtr`.
- `resetBottomList:` rearrange the states in `BottomList` & `NonBottomList`.

An LTS is a structure containing

[Data Members]

- `InitStateId:` an integer referring to the initial state of LTS.
- `States:` the state set of LTS.
- `Trans:` the transition set of LTS.
- `Actions:` the action set of LTS.

A state is a structure containing

[Data Members]
- `Id:` an integer which is its representative.
- `BlockPtr:` a pointer reference to the block it belongs to.
- `Mark:` a Boolean variable.
- `InTrans:` a set of incoming transition pointers.
- `OutTrans:` a set of outgoing transitions pointers.
- `InTauTrans:` a set of incoming τ-transitions pointers.
- `OutTauTrans:` a set of outgoing τ-transitions pointers.

[Member Functions]
- `setMark:` set `Mark` to true or false.
- `isMarked:` return the value of `Mark`.

A transition is a structure containing

[Data Members]
- `SrcState:` a pointer to its source state.
- `Action:` an integer referring to an action.
- `TrgState:` a pointer to its target state.

Note that the data types of the elements in `BottomList`, `NonBottomList`, `TransitionList`, `BlockList` etc are pointers to objects of states, transitions, or blocks. In our implementation, we use pointers to objects for several reasons. (1) The element inserted into a container of STL is duplicated. Using the data type of pointers can avoid wasting unnecessary memory space; (2) During the minimization process,

we often need to merge states and remove τ-transitions. The memory space of an object can be easily released by its pointer. Therefore we and save more space immediately for composing; (3) If we need to maintain several copies of an object, each copy only requires the size of a pointer type. The use of memory may not grow up rapidly for each replica.

The reason why we keep so many transition references in a state structure is for quickly accessing to a certain transition related to a state. Here are some examples. We need to gather incoming transitions (`InTrans`) and outgoing transitions (`OutTrans`) of a block which have to be preserved for merging the block into one state. For finding strong connected components with respect to τ-transitions, the back tracing for incoming τ-transitions (`InTauTrans`) of a state is required for doing reversed depth first search. The outgoing τ-transitions (`OutTauTrans`) are also needed to achieve topological sorting. These related works will be specified in later sections.

In addition, the list structures, such like state lists, transition list, block lists etc, are used frequently in the program. We use the hash-based structure (e.g., hashed map, hashed multimap, hashed set, etc) to improve the efficiency of accessing to elements in a list, if the sequence of inserting elements into the list is not concerned.

3.3 Preprocess

```
Let statelist be a global variable;
Let SCC be a global variable;

 1: void MergeTauCycle( LTS ) {
 2:     Let statelist be an empty list;
 3:     for all s in LTS.States {
 4:         if( ! s.isMarked() )
 5:             DFS( s );
 6:     }
 7:     for all s in statelist {
 8:         if( ! s.isMarked() ) {
 9:             Create an SCC structure;
10:             rDFS( s );
11:             MergeStates( SCC );
12:         }
13:     }
14: }

15: void DFS( s ) {
16:     s.setMark(true);
17:     for all t in s.OutTauTrans {
18:         trg = t.TrgState;
19:         if( ! trg.isMarked() )
20:             DFS( trg );
21:     }
22:     statelist.addfirst( s );
23: }

24: void rDFS( s ) {
25:     s.setMark(true);
26:     for all t in s.InTauTrans {
27:         src = t.SrcState;
28:         if( ! src.isMarked() )
29:             DFS( src );
30:     }
31:     SCC.addfirst( s );
32: }
```

Figure 3.2. The algorithm of merging τ-cycles.

The purpose of the preprocessing is to find those τ-cycles in the LTS, and merge each τ-cycle into one state with a self-τ-cycle. Thus, a graph without cycles of τ-transitions is obtained. The algorithm of finding *strong connected components* (SCC for short) introduced in [31][32] is applied to solve this problem.

The algorithm of **`MergeTauCycle`** consists of two phases, depth first search (function **`DFS`**) and reversed depth first search (function **`rDFS`**). The resulting sequence of *`statelist`* after calling **`DFS`** can not be mistaken. At line 22 in Figure 3.2, the position of inserting `s` into *`statelist`* is the key point which affects the sequence of states in *`statelist`* sorted topologically. For more detail about finding SCC in a graph, one can finds the answer in [32].

The SCC structure mentioned at line 9 is a kind of state list gathering those states formed a τ-cycle. The function of **`MergeStates`** is used for merging these state into one state and preserve necessary transitions. The details of **`MergeStates`** is shown in section 3.8.

3.4 Main Procedure

```
 1: Let StableBlockList be an empty block list;
 2: Let UnstableBlockList be an empty block list;
 3: Let BlockList be an empty block list;

 4: void BranchBisim( LTS ) {
 5:    Let B be a block initialized from LTS;
 6:    UnstableBlockList.addfirst( B );
 7:    while UnstableBlockList is not empty {
 8:       B′ = UnstableBlockList.first();
 9:       UnstableBlockList.delete( B′ );
10:       StableBlockList.addfirst( B′ );
11:       CheckUnstableBlock( B′ );
12:    }
13:    for all b in StableBlockList {
14:       MergeStates( b );
15:    }
16: }
```

Figure 3.3. The main procedure of Branching Bisimulation Minimization.

There are three *global* variables through out the algorithm. The first one is `StableBlockList`, which contains the blocks with respect to which each blocks of the current partition is stable. The second one is `UnstableBlockList`, which contains the blocks with respect to which the instability notion has to be checked for each blocks of the partition. The last one is `BlockList`, which contains those potentially unstable blocks gathered by the function **`Unstable`** (see section 3.6). The

first and the third are initially empty, and the second contains the unique block with all states and transitions of *LTS* to be the initial partition (line 6).

From line 7 to line 12, the main loop takes a block *B′* as a *splitter* from *UnstableBlockList* and put it *temporally* into *StableBlockList*. Then it calls the function **CheckUnstableBlock** to check the instability notion of other blocks in the current partition with respect to *B′*. Till *UnstableBlockList* is empty which means the current partition is stable, then we can merge those blocks in *StableBlockList* for minimization (line 13~15). The detail of merging blocks is described in section 3.8.

A more detail discussion on initializing a block is given in the following subsections.

3.4.1 Initialization of a Block

At line 5 in Figure 3.3, a block *B* is initialized from the input *LTS* object. First we set up *BottomList* and *TransitionList* of *B* with *LTS.States* and *LTS.Trans* respectively. Then we do the following three steps to initialize the status of block *B*.

1. *B*.resetMark(): clears all *mark*s of states in BottomList;
2. *B*.resetBlockPtr(): sets *BlockPtr*s of states to refer to *B*;
3. *B*.resetBottomList(): decides whether each state belongs to *BottomList* or *NonBottomList*.

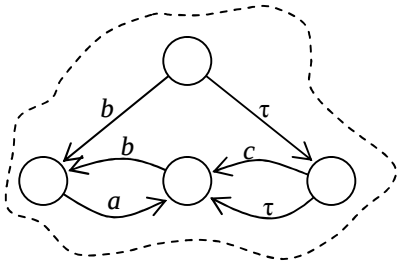
The first two steps are easy to implement. In the following subsection, we show our algorithm for deciding bottom states and non-bottom states corresponding to the definition of ***bottom***(*B*) (see section 2.5).

3.4.2 Deciding BottomList and NonBottomList

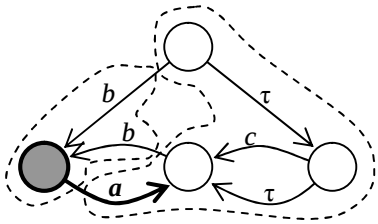
```
 1: void Block::resetBottomList() {
 2:     for all s in BottomList {
 3:         for all t in s.OutTrans {
 4:             if t.Action==τ and t.TrgState≠s
                                and t.TrgState is in this block {
 5:                 BottomList.delete( s );
 6:                 NonBottomList.addfirst( s );
 7:             }
 8:         }
 9:     }
10:     for all s in NonBottomList {
11:         bool becomeBottom = true;
12:         for all t in s.OutTrans {
13:             if t.Action==τ and t.TrgState≠s
                                and t.TrgState is in this block {
14:                 becomeBottom = false;
15:                 break;
16:             }
17:         }
18:         if( becomeBottom ) {
19:             NonBottomList.delete( s );
20:             BottomList.addfirst( s );
21:             Move all elements in StableBlockList to UnstableBlockList;
22:         }
23:     }
24:     TopologicalSort( NonBottomList );
25: }
```

Figure 3.4. The algorithm for deciding bottom states and non-bottom states.

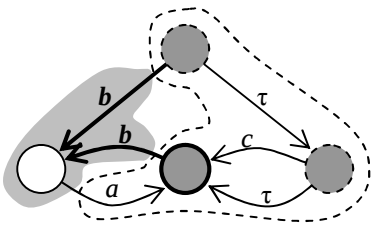
From line 2 to line 9, we check which state in `BottomList` has τ-transitions (except self-τ-cycle), and move it to `NonBottomList`.



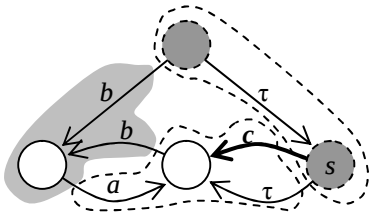
(1) The initial partition.



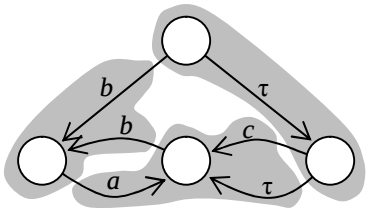
(2) Check unstable w.r.t. *a*.



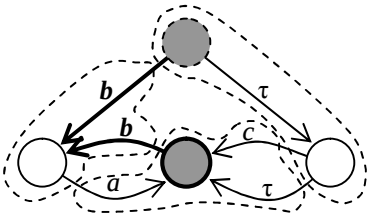
(3) Check unstable w.r.t. *b*.



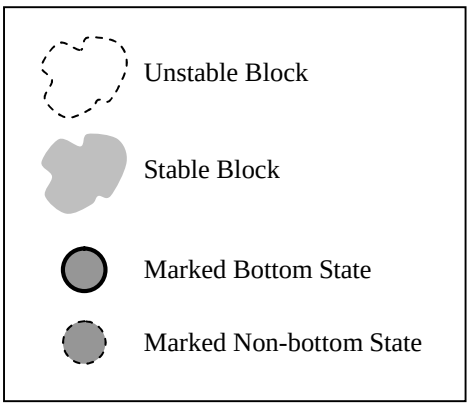
(4) Check unstable w.r.t. *c*.
Non-bottom *s* becomes bottom.



(5.1) Without appending stable block list to unstable block list, and get wrong result.



(5.2) Append stable block list to unstable block list, and check instability again.



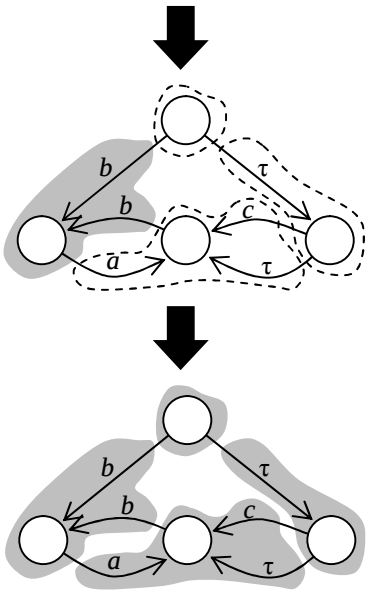


Figure 3.5. A counterexample on deciding non-bottom states.

From line 10 to line 23, we check which state in *NonBottomList* has no τ-transitions (except self-τ-cycle), and move it to *BottomList*. Moreover, if a non-bottom state becomes a bottom state, then we append *StableBlockList* to *UnstableBlockList* and make *StableBlockList* empty (lines 18~22) [26]. This operation is not shown clearly in the algorithm presented by Bouali [27]. There is a counterexample in Figure 3.5, if the operation is disregarded.

At the end of the procedure, the states in *NonBottomList* have to be sorted topologically. This improves the efficiency of finding successive states of $s \stackrel{\tau}{\Longrightarrow} s'$ in the function **SplitNonBottom** (see section 3.7) in $O(m)$ time complexity. In the next subsection we show the algorithm of topological sorting [31][32].

3.4.3 Topological Sorting on NonBottomList

```
1: Let resultlist be an global variable;
2: void TopologicalSort( statelist ) {
3:    Let resultlist be an empty list;
4:    for all s in statelist {
5:       if( ! s.isMarked() ) {
6:          DFS( s );
7:       }
8:    }
9: }
10: void DFS( s ) {
11:    s.setMark(true);
12:    for all t in s.OutTauTrans {
13:       trg = t.TrgState;
14:       if( ! trg.isMarked() )
15:          DFS( trg );
16:    }
17:    resultlist.addlast( s );
18: }
```

Figure 3.6. The algorithm of topological sorting.

This procedure sorts the input state list topologically with respect to τ-transitions. One may identify that the function **DFS** is similar to the one used by

MergeTauCycle in section 3.3. But there is slightly different at line 17 from the line 22 in Figure 3.2. This makes the resulting sequence of states be like $0 \leftarrow 1 \leftarrow 2 \leftarrow 3$, which means the successive states of each state is placed in front of it. So it is convenient to traverse the list from the beginning to check whether successive states is marked or not in $O(m)$ time complexity.

One may try to integrate both **DFS** functions in **MergeTauCycle** and **TopologicalSort** into one common function. But the direction of traversing a list must be considered carefully.

3.5 Check Unstable Blocks

```
 1: void CheckUnstableBlock( B ) {
 2:     for all a in sort(B) and StableBlockList.first()==B {
 3:         Unstable( a, B );
 4:         Refine( B );
 5:     }
 6: }

 7: set sort( B ) {
 8:     Let act be an empty set;
 9:     for all t in B.TransitionList {
10:         if( t.Action≠τ or t.SrcState is not in B )
11:             act.add( t.Action );
12:     }
13:     return act;
14: }
```

Figure 3.7. The algorithm of checking unstable blocks.

In order to check whether other blocks are required to be split with respect to those transitions ending in B, **CheckUnstableBlock** goes through all actions except τ inside B with a loop. We use a formal definition of ***sort***(B) to specify these

actions.

$$\textbf{\textit{sort}}(B) = \{\, a \in A \mid \exists s \in B \ni s' \xrightarrow{a} s \wedge (a = \tau \Rightarrow s' \notin B) \,\}$$

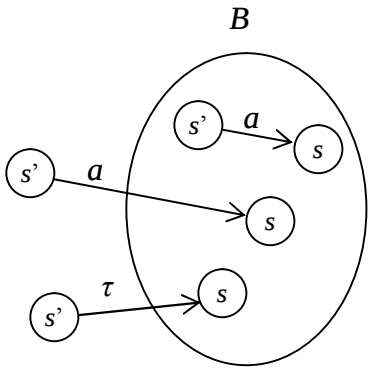


Figure 3.8. The notion of sort(B).

We depict the notion of ***sort***(*B*) in Figure 3.8. For more detailed description, ***sort***(*B*) collects the actions of incoming transitions from other blocks or those of non-τ-transitions in *B* itself.

In addition to ***sort***(*B*), we also consider the condition of the first element of `StableBlockList` which is added by main procedure as a *splitter*. If *B* splits itself into *B1* and *B2* (as `StableBlockList.first()==B` is false at line 2) in the refinement step, then we exit the loop and go back to main loop to choose another splitter. The reason to stop checking more transitions ending in *B* is because they must again be checked for *B1* and *B2* later [26].

3.6 Obtain Unstable Blocks by Instability Notion

```
1: void Unstable( a, B ) {
2:    for all t in B.TransitionList {
```

```
 3:        if t is a τ transition in B then continue;
 4:        if( t.Action == a ) {
 5:           t.SrcState.setMark(true);
 6:           if t.SrcState.BlockPtr is not in BlockList
 7:              BlockList.addfirst( t.SrcState.BlockPtr );
 8:        }
 9:     }
10: }
```

Figure 3.9. The algorithm of obtaining potentially unstable blocks.

This procedure collects those *potentially* unstable blocks which have transitions ending in B via the action a into `BlockList` (line 7), and marks the source states of these related transitions (line 5).

However, there is a particular situation if we want to check the instability with τ action, i.e. $a = \tau$. We can not mark any source state of τ transitions arbitrarily. The τ transitions inside B must be ignored (line 3) to meet the correct definition of the instability notion.

Furthermore, there are two things different from the original procedure in [27]. The first is that we ensure an empty `BlockList` used here by resetting it at the end of the procedure **`Refine`**. The second is that we do not do extended marking at the end of ours. The reason is because this must again be done by the following refinement step if B is also in `BlockList`.

3.7 Refine Unstable Blocks

```
1: void Refine( B ) {
2:    for all b in BlockList and StableBlockList.first()==B {
3:       if NOT all bottom states of B are marked
```

```
 4:           Split( b );
 5:    }
 6:    for all b in BlockList {
 7:       b.resetMark();
 8:    }
 9:    BlockList.clear();
10: }
```

Figure 3.10. The algorithm of refining unstable blocks.

In the function **Refine**, we go through the *BlockList* to check the marks of bottom states in each block according to the instability notion described at **Definition 2.2**. We identify a block as an actually unstable one, if the bottom states of a block are *partially* marked (line 3). Then the block is separated by **Split** function shown below.

After leaving the loop, it is important to reset the *BlockList* (from line 6 to line 9). Resetting states' mark of those blocks in *BlockList* is not mentioned in [27] clearly. If the marks are not reset, this may affect the following refinement step according to marks.

```
 1: void Split( B ) {
 2:    Let B1 be a reference to B;
 3:    Let B2 be an empty block;
 4:
 5:    SplitBottom( B1, B2 );
 6:    SplitNonBottom( B1, B2 );
 7:    SplitTransition( B1, B2 );
 8:
 9:    if B1 is empty {
10:       Restore B2 back to B1;
11:       Release the memory space of B2;
12:       B1.resetMark();
13:       B1.resetBottomList();
14:    }else {
15:       if B1 is in StableBlockList {
16:          StableBlockList.delete( B1 );
17:          UnstableBlockList.addfirst( B1 );
```

```
18:         }
19:         B1.resetBottomList();
20:         B2.resetMark();
21:         B2.resetBlockPtr();
22:         B2.resetBottomList();
23:         UnstableBlockList.addfirst( B2 );
24:     }
25: }
```

Figure 3.11. The Split function.

The **Split** function tries to separate block *B* into *B1* and *B2*. There are two conditions after splitting states and transitions (lines 5~7).

One is that states and transitions of *B1* are totally moved to *B2* so that *B1* becomes empty. This happens because the original block *B* has no bottom states, and its non-bottom states are all marked. In order to keep *B1* in whatever *StableBlockList* or *UnstableBlockList*, we restore all states and transitions in *B2* back to *B1*, and reset the status of *B1* (lines 10~13).

The other condition is that block *B* is actually separated into two distinct blocks, *B1* and *B2*. And it is required to move *B1* and *B2* to *UnstableBlockList* to check their instabilities again.

We give the algorithms of splitting states and transitions below.

```
1: void SplitBottom( B1, B2 ) {
2:     for all s in B1.BottomList {
3:         if( s.isMarked() ) {
4:             B1.BottomList.delete( s );
5:             B2.BottomList.addfirst( s );
6:         }
7:     }
8: }
```

Figure 3.12. The SplitBottom function.

SplitBottom separates those marked states in *BottomList* from *B1* to *BottomList* of *B2*.

```
 1: void SplitNonBottom( B1, B2 ) {
 2:    for all s in B1.NonBottomList {
 3:       if( s.isMarked() ) {
 4:          B1.NonBottomList.delete( s );
 5:          B2.NonBottomList.addfirst( s );
 6:       }else {
 7:          for all t in s.OutTauTrans {
 8:             if t.TrgState is in B1 and t.TrgState.isMarked() {
 9:                s.setMark(true);
10:                B1.NonBottomList.delete( s );
11:                B2.NonBottomList.addfirst( s );
12:             }
13:          }
14:       }
15:    }
16: }
```

Figure 3.13. The SplitNonBottom function.

SplitNonBottom separates not only those marked states in *NonBottomList* but also those states which can reach to marked states via τ actions, i.e. $s \stackrel{\tau}{\Longrightarrow} s'$. This can be done efficiently with topological sorting on *NonBottomList* in advance (see section 3.4.3).

```
1: void SplitTransition( B1, B2 ) {
2:    for all t in B1.TransitionList {
3:       if( t.TrgState.isMarked() ) {
4:          B1.TransitionList.delete( t );
5:          B2.TransitionList.addfirst( t );
6:       }
7:    }
8: }
```

Figure 3.14. The SplitTransition function.

SplitTransition moves those transitions whose *target states* are marked from *B1* to *B2*, so this procedure must be called after **SplitNonBottom** which does extended marking to non-bottom states.

3.8 Merge States

The approach of merging states plays an important role in the whole program. It affects the outcome of minimization directly. Even though the former algorithms are totally implemented correctly, without merging states we can not obtain the final result.

There are two situations where some specific states have to be merged. One is to merge those states formed a τ-cycle. The other is to merge those states in a block. The basic procedure of them is identical. Figure 3.15 shows the common algorithm.

```
 1: Let UnwantedTrans be an empty list of transitions;
 2: Let SelfAction be an empty action set;
 3: Let OutTrans be an empty list of transitions;
 4: Let InTrans be an empty list of transitions;

 5: void MergeStates ( statelist ) {
 6:     Let id be one of state ids in statelist;
 7:     Inspect( statelist );
 8:     Remove UnwantedTrans;
 9:     Remove states in statelist;
10:     Create a new state S with id;
11:     Add self cycle with SelfAction to S;
12:     Add OutTrans to S;
13:     Add InTrans to S;
14: }
```

Figure 3.15. The common algorithm of merging states.

There are four global variables through out the algorithm (line 1~4). *UnwantedTrans* collects those transitions required to be deleted from LTS. *SelfAction* collects those actions had to be remained to form self cycles of the new state. *OutTrans* and *InTrans* are those transitions preserved to become outgoing transitions and incoming transitions of the new state respectively. These four global objects are maintained by **Inspect** function to gather those elements needed.

At line 6, we choose one state id from *statelist* (e.g., *id* = *statelist*.first( );) rather than generate an arbitrary one. We use this method to avoid matters of maintaining unused id numbers. The **Inspect** function at line 7 will be described in the following subsections according to specific conditions.

At line 8, those unwanted transitions in *UnwantedTrans* list are removed from related states, which are not in *statelist*, and *LTS.Trans*. Then we release those original states in *statelist* at line 9, and create another new state at line 10. From line 11 to line 13, the related transitions of the new state are connected.

The basic concept of this algorithm is to collect those transitions still required on the new state and remove those unwanted. More clearly, those required transitions on the new state are classified into three groups:

1. **Incoming transitions:** from those transitions coming from other states not in *statelist*;

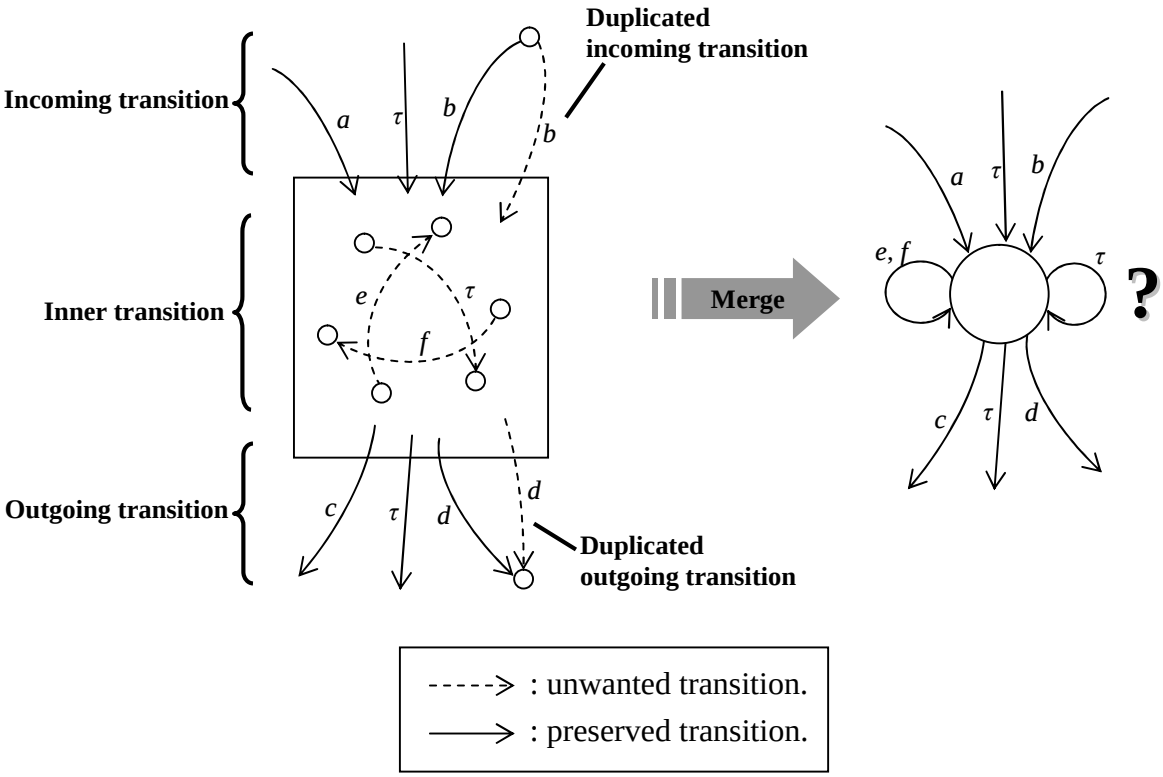


Figure 3.16. The basic concept of merging states.

2. **Self cycles:** with actions of inner transitions between states in `statelist`; and
3. **Outgoing transitions:** from those transitions going to other states not in `statelist`.

And those unwanted transitions are also gathered from three kinds of transitions:

1. **Inner transitions:** whose source state and target state are both in the original `statelist`;
2. **Duplicated incoming transitions:** whose source states (not in `statelist`) and actions are the same, but target states are referring to distinct states in `statelist`; and
3. **Duplicated outgoing transitions:** whose target states (not in `statelist`) and actions are the same, but source states are referring to distinct states in `statelist`.

We show the mapping on transitions between original states and the new state in

Figure 3.16. Notice that there is no dispute about incoming and outgoing τ-transitions

which are directly left for the new state. But it is not quite trivial for inner τ-transitions. This is the great distinction between merging states of a τ-cycle and a block – the decision of whether the new state holds a self-τ-cycle or not. Our solution is presented in the following subsections.

3.8.1 Merging States formed a τ-cycle

```
 1: void InspectForTauCycle( statelist ) {
 2:    for all s in statelist {
 3:       for all t in s.OutTrans {
 4:          if t.TrgState is not in statelist {
 5:             OutTrans.add( t );
 6:          }else {
 7:             SelfAction.add( t.Action );
 8:             UnwantedTrans.add( t );
 9:          }
10:       }
11:       for all t in s.InTrans {
12:          if t.SrcState is not in statelist {
13:             InTrans.add( t );
14:          }
15:       }
16:    }
17: }
```

Figure 3.17. The algorithm of InspectForTauCycle.

It is trivial to always keep a self-τ-cycle for the new state in the condition of merging states formed a τ-cycle. Therefore, in the algorithm we only have to gather those actions (including τ and non-τ) of inner transitions to become self cycles of the new state (line 7) .

3.8.2 Merging States in a Block

```
 1: void InspectForBlock( statelist ) {
 2:     for all s in statelist {
 3:         bool hasInnerTran = false;
 4:         bool hasInnerNonTau = false;
 5:         bool hasSelfTau = false;
 6:         for all t in s.OutTrans {
 7:             if t.TrgState is not in statelist {
 8:                 OutTrans.add( t );
 9:             }else if( t.TrgState == t.SrcState ) {
10:                 if( t.Action ≠ τ )
11:                     SelfAction.add( t.Action );
12:                 else
13:                     hasSelfTau = true;
14:                 UnwantedTrans.add( t );
15:             }else {
16:                 if( t.Action ≠ τ ) {
17:                     SelfAction.add( t.Action );
18:                     hasInnerNonTau = true;
19:                 }
20:                 hasInnerTran = true;
21:                 UnwantedTrans.add( t );
22:             }
23:         }
24:         if( hasSelfTau and (hasInnerNonTau or !hasInnerTran) )
25:             SelfAction.add( τ );
26:         for all t in s.InTrans {
27:             if t.SrcState is not in statelist {
28:                 InTrans.add( t );
29:             }
30:         }
31:     }
32: }
```

Figure 3.18. The algorithm of InspectForBlock.

According to the comparison with the result of FC2Tools [35] and plenty of try-and-error tests, we found that there are two conditions for the new state owning a self-τ-cycle: if there is a state with a self-τ-cycle in the block and either (1) it has no successor or (2) it has inner non-τ-transition. The decision is made by using three Boolean variables: *hasSelfTau*, *hasInnerNonTau*, and *hasInnerTran*, and determined at line 24.

An example of condition (1) is shown in Figure 3.19. As long as there is one (or more than one) terminal state with a self-τ-cycle (like the state in gray) in the block, a self-τ-cycle will be required for the new state.

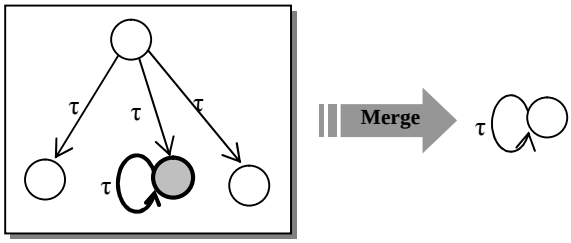


Figure 3.19. Condition (1) of deciding self-τ-cycle on merging a block.

An example of condition (2) is shown in Figure 3.20. There exists the gray state with a self-τ-cycle having inner transition labeled by *a* (any *non-τ action*), so the new state must also contain a self-τ-cycle.

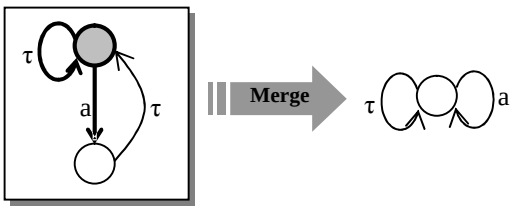


Figure 3.20. Condition (2) of deciding self-τ-cycle on merging a block.

The complexity of the **`Inspect`** function is $O(m+n)$, because for each state we only check its outgoing transitions to whatever states and incoming transitions from states not in `statelist`. This assures that each transition belonging to states in `statelist` is only checked once.