Chapter 4. On-the-fly Minimization

In this chapter, we demonstrate an algorithm for the on-the-fly branching bisimulation minimization in compositional analysis. As we described in section 1.2, in order to automatically obtain a simpler interface process for further composing, minimizing the state space is necessary. However, it is still possible to fail if the state space of a subsystem is already too excessive to be handled by the limited memory resources. Typically, in this situation, the O.S. begins thrashing due to the page replacement.

Therefore, minimizing the partially explored state space during parallel composition is attractive for the occasion. This improves the memory usage on composing large subsystems by releasing unwanted state space and reserving it for further enumeration in progress. This approach is typically known as the "*on-the-fly*."

We make our best efforts in discovering the principle of on-the-fly minimization according to the instability notion of branching bisimulation. We hope to minimize the partially explored state space and obtain the same result as it is minimized at once. In the following sections, first we illustrate the principle of how to make the idea work properly, and then present the *interface procedure* by which the branching bisimulation minimization can be invoked easily in the process of parallel composition without

modifying the code of minimization. At last we show a general algorithm of compositional analysis integrated with minimization.

4.1 The Principle behind the On-the-fly Branching Bisimulation

Overall, bisimulation is defined in terms of recursive relation. So, in principle, the equivalence can only be determined until the whole state space is explored. So, during parallel composition, when a partially explored state space is minimized, we need to make sure we do not destroy the original behaviors.

The difference between on-the-fly and off-line minimizations is that the former minimize a whole system in several rounds, and the latter at once. More clearly, the object for one-the-fly minimization is a partially composed LTS whose state space remains to be explored. Therefore, we have to clarify the meaning of the partially composed LTS. We first give a general definition of *a partial LTS* below.

Definition 4.1. A Partial LTS

Given a whole system $L=<S, A, \rightarrow, q>$, $L'=<S', A', \rightarrow', q'>$ is *a partial LTS* with respect to *L iff*

$q' = q \ \wedge \ S' \subseteq S \ \wedge \ A' \subseteq A \ \wedge \ \rightarrow' \subseteq S' \times A' \times S' \cap \rightarrow$ such that

$\forall s \in S', \ \exists s_i \in S', \exists a_i \in A', q'=s_0 \xrightarrow{a_0} s_1 \cdots s_{n-1} \xrightarrow{a_{n-1}} s_n=s$, where $0 \le i \le n$, $n \in \mathbf{N}$.

□

We use $L' \subseteq_p L$ to denote a partial LTS L' which is a directed graph with partial traces from the same root of L. We show an example in Figure 4.1 to illustrate that L' is a partial LTS w.r.t. L. Each state (in gray) in L' can be reached from the root.

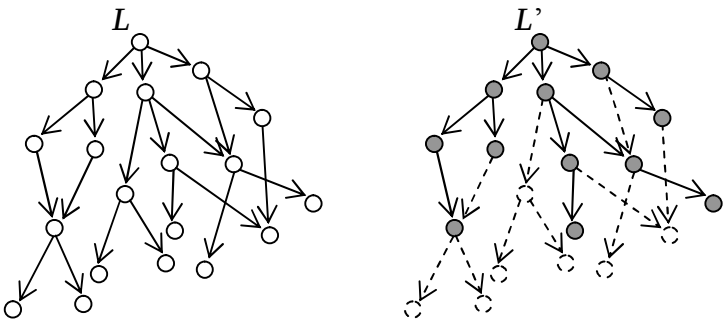


Figure 4.1. L' is a partial LTS w.r.t. L.

If we take the definition of the partial LTS to be the object for minimization, this means that we can do minimization at any time whatever the structure of current composed LTS is built like. But this is too simple to regard the instability notion.

By reviewing the instability notion in **Definition 2.2**, we make an induction of that the significant factor of deciding whether the states in a block B should be separated by a splitter B' or not is those specific transitions ending in the splitter. In other words, the condition of that a block is stable or unstable depends on those transitions outgoing to other blocks via any actions (including τ).

We can also perceive this notion from the code of **`Unstable`** function in Figure 3.9. At line 11, it marks the source state of a transition which comes from a block may be considered unstable and required to be split.

Therefore, based on the notion described above, when we need to minimize a partially composed LTS with the guaranteed instability notion for the entire system, the outgoing transitions of each non-terminal state in the partially composed LTS must be totally enumerated for sure. This assures that the instability of a block can be totally checked with its outgoing transitions of current enumerated states. Hence, a formal definition of a partially composed LTS with the guaranteed instability notion is given below.

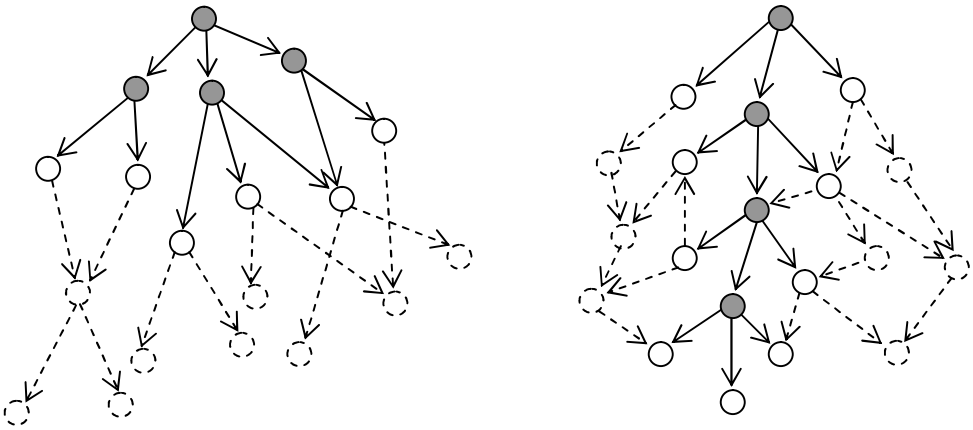
Definition 4.2. A Partially Composed LTS

Given a whole system $L=<S, A, \rightarrow, q>$, $L'=<S', A', \rightarrow', q'>$ is *a partially composed LTS* with respect to L *iff*

$L' \subseteq_p L \ \wedge \ \forall s \in S', \boldsymbol{outT}_{L'}(s) \neq f \ \wedge \ \boldsymbol{outT}_{L'}(s) = \boldsymbol{outT}_{L}(s)$, where
$\boldsymbol{outT}_{L}(s) = \{ (s, a, s') \in \rightarrow \mid L=<S, A, \rightarrow, q>, a \in A, s' \in S \}$ □

Definition 4.2 shows that L' is a partial LTS w.r.t. L, and each non-terminal state (denoted as $\boldsymbol{outT}_{L'}(s) \neq f$) in L' has the same outgoing transitions in L. The $\boldsymbol{outT}_{L}(s)$ denotes those outgoing transitions of s in S of L. And $\boldsymbol{outT}_{L'}(s) = \boldsymbol{outT}_{L}(s)$ means that all outgoing transitions of each non-terminal state in L' are totally constructed.

We briefly illustrate this notion in Figure 4.2. No matter what algorithms applied to enumerating the state space, the outgoing transitions of all non-terminal state of the partially composed LTS must be built up totally by composing.



(a) By Breadth First Search (b) By Depth First Search

Figure 4.2. A partially composed LTS.

In either (a) or (b) of Figure 4.2, there are 4 of the enumerated states are non-terminal (states in gray) in current enumerated state space, and their outgoing transitions are all enumerated totally. Therefore, these two LTS fit the condition to be minimized with the guaranteed instability notion for the entire system.

Now we explain the reason why a partially composed LTS of **Definition 4.2** can guarantee the instability notion. In order to maintain the result of on-the-fly minimization corresponds to that of off-line minimization, the on-the-fly algorithm has to accept two propositions. (Π denotes the stable partition of off-line minimization.)

1. Those states in the same block of Π are eventually merged.

2. Those states in distinct blocks of Π always can not be merged.

We introduce a *dummy* on-the-fly minimization algorithm to explain the first proposition. The dummy algorithm separates each state into one block regardless the

instability notion in intermediate round and does the real minimization after enumerating the overall state space. No doubt those states in the same block of Π should be merged at last. This implies that if the refinement of splitting blocks is applied adequately and sufficiently in each intermediate round such that states are separated in a rigorous way, then those states should be merged will eventually be placed in the same block at the last refinement. Therefore, we can perceive that the first proposition is always true, if the second is accepted.

We explain the second proposition based on the instability notion. A block is considered potentially unstable because some states in it are marked by the splitter block. More clearly, a block may be required to be split due to those transitions reaching to other blocks via any actions. Therefore, if we want to fully check whether a state should be separated from another, those outgoing transitions of the state is the major cause which leads to different behaviors from another. This is why we introduce the notion that all outgoing transitions of each non-terminal state must be constructed in **Definition 4.2**. This guarantees that each non-terminal state in a partially composed LTS can be fully identified with regard to the instability notion and always be separated from those intended states. Consequently, the definition of partially composed LTS accepts the two propositions.

In the following content, minimizing a partially composed LTS is guaranteed the

instability notion for the whole system. But before doing minimization, we also have to merge τ-cycles first and prepare the initial partition at beginning. We introduce the interface procedure for this purpose in the following section.

4.2 Integration Interface Procedure

In this section, we discuss the procedure after obtaining a partially composed LTS from intermediate composition. As the same with general minimization, the partially composed LTS is required the preprocessing of removing τ-cycles at first. However, there is slightly different in preparing the initial partition for branching bisimulation minimization.

We start with the problem of merging τ-cycles in a partially composed LTS. Considering the example in Figure 4.3, these 4 states are eventually merged into one. Although these states may be split from others due to transitions labeled by a_1, a_2, and a_3, and not be merged at once, a τ-cycle will still be constructed while outgoing transitions are all available.

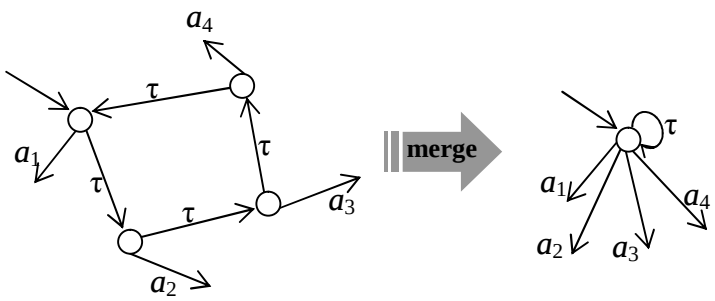


Figure 4.3. Merging τ-cycles in a partially composed LTS.

Next we consider the case of preparing the initial partition. We have to notice that some terminal states (white circles with solid line in Figure 4.2) in a partially composed LTS will hold their own outgoing transitions which may affect the overall refinement in later minimization. So we can not directly take the partially composed LTS as the unique initial partition. This will result in merging those terminal states into one state. Therefore, we have to rescue them out to make each of them become a single block. Furthermore, they play an import role in maintaining the instability of the current partition. They are all splitters to other blocks. Via incoming transitions of the splitter, we can mark their source states to find out potentially unstable blocks.

Here we call the block, which contains only one terminal state and its incoming transitions in a partially composed LTS, a ***splitter-block***. And the block made up of other states is called the ***main-block***. Therefore, the initial partition is consisted of one ***main-block*** and some ***splitter-blocks***. We show this notion in Figure 4.4.

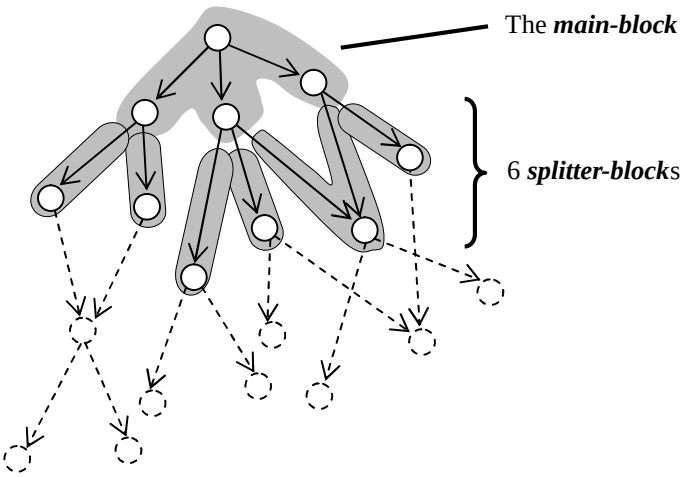


Figure 4.4. The main-block and splitter-blocks.

According to the notion of splitter-blocks, they are those terminal states in a partially composed LTS. One may prepare the initial partition as we described above for intermediate minimization. But this may fail on the final minimization which must take the overall enumerated state space as the unique initial partition. Moreover, there are sometimes states which actually do not have outgoing transitions in the progress of composing. Although they do not affect the result of refinement in the intermediate minimization, we can merge them into one state earlier to save more memory in use. Therefore, taking non-terminal states to be the splitter-blocks is not always adequate. We tackle this problem by using a state list to specify those states which should become the splitter-blocks. The detail implementation of deciding the state list will be discussed later in section 4.3.

The last issue we thought about is that – Is it necessary to minimize the whole LTS in each round? Can we ignore those states which are split already in previous rounds and leave them out of the initial partitions in later rounds? The answer is definitely no, because we can not assure that there is no state whose behavior is equivalent to the existed one in later enumerations. It is possible for a previous state to be merged with a later one, so minimizing the overall state space of a partially composed LTS is always required.

Finally we show the interface procedure in Figure 4.5.

```
 1: Let UnstableBlockList be an empty block list;
 2: Let composedLTS be a partially composed LTS;

 3: void Minimization( splitterlist ) {
 4:    MergeTauCycle( composedLTS );
 5:    PrepareUnstableBlockList( splitterlist );
 6:    BranchBisim( composedLTS );
 7: }

 8: void PrepareUnstableBlockList( splitterlist ) {
 9:    Let MainBlock be an empty block;
10:    for all s in composedLTS.States {
11:       if s is in splitterlist {
12:          Create an empty block b; // the splitter-block
13:          b.BottomList.add( s );
14:          Add all s.InTrans into b.TransitionList;
15:          UnstableBlockList.add( b );
16:       }else {
17:          MainBlock.BottomList.add( s );
18:          Add all s.InTrans into MainBlock.TransitionList;
19:       }
20:    }
21:    MainBlock.resetMark();
22:    MainBlock.resetBlockPtr();
23:    MainBlock.resetBottomList();
24:    UnstableBlockList.add( MainBlock );
25: }
```

Figure 4.5. The interface procedure for on-the-fly minimization.

We wrapped the algorithm of minimization into one interface procedure named **Minimization** for the convenient invocation in the progress of composing. Two global variables are used in the interface procedure. The first is *UnstableBlockList* which is used to contain the initial partition for the algorithm of branching bisimulation minimization. So the main procedure **BranchBisim** in Figure 3.3 needs to be modified slightly. The line 5 and 6 in Figure 3.3 must be taken away for proper adaptation. And *composedLTS* is the second global variable which refers to the partially composed LTS. The procedure **Minimization** accepts an input parameter, *splitterlist*, for the use of preparing the initial partition. In the

function **PrepareUnstableBlockList**, states in *composedLTS.States* are separated into the main-block and the splitter-blocks according to the *splitterlist*. Those states in *splitterlist* are become a splitter-block with its incoming transitions. Other states out of *splitterlist* are gathered into *MainBlock* also with their incoming transitions. And at last the status of *MainBlock* must be set appropriately for starting the minimization.

4.3 Sample Algorithm of Integrating Minimization with Composition

```
 1: list<lts> InputLtsList;
 2: lts composedLTS;

 3: void BFS_Compose() {
 4:    Let InitState be a state with id of concatenation of each
    InitStateId of InputLtsList;
 5:    composedLTS.States.add( InitState );
 6:    composedLTS.InitStateId = InitState.Id;
 7:    Queue.addtail( InitState );
 8:    do {
 9:       CurState = Queue.delhead();
10:       for all s be successors of CurState {
11:          Let t be a transition of CurState --a--> s;
12:          composedLTS.States.add( s );
13:          composedLTS.Trans.add( t );
14:          if s is new created {
15:             Queue.addtail( s );
16:          }
17:       }
18:       if minimization is required
19:          Minimize( Queue );
20:    }while( ! Queue.empty() );
21:    Minimize( Queue );
22: }
```

Figure 4.6. The algorithm of integrating minimization with composition.

We demonstrate a sample algorithm of composing by breadth first search with on-the-fly minimization, and describe how to keep the state list for preparing the initial partition for minimization.

The first two global variables at line 1 and 2 are needed. *InputLtsList* contains LTS objects of those original processes required composing. And *composedLTS* is the resulting LTS object.

From line 4 to line 6, we first create the initial state of *composedLTS* from those initial states of original processes, and add it to *Queue* to start enumerating its successive traces.

From line 8 to line 20, there is a loop for each enumerated state. In the loop, we construct the successive traces containing outgoing transitions and target states for each reachable state, and then put those target states into *Queue* for subsequent iterations until the *Queue* is empty. This is a general breadth-first-search algorithm for composing processes. Here we have to notice that if some states in the partially composed LTS were merged, then the target states of those later constructed outgoing transitions reaching states merged have to be redirected to the merged states correctly.

Next we consider the case of invoking minimization procedure. As we described in previous section, only the partially composed LTS with outgoing transitions of

non-terminal states all constructed is adequate for minimization. Therefore, we add the invocation of minimization after all successors of a state are enumerated (from line 10 to line 17). And the last round for minimizing the overall state space is also required after the whole composition process (line 21).

The last and the most important key point for preparing initial partition is the *Queue*. It contains those new generated states which have no outgoing transitions yet. These states in *Queue* are the most preferable for becoming the *splitter-blocks* in an initial partition. Moreover, with the use of *Queue* we can prevent those states which actually have no outgoing transitions to become splitter-blocks in subsequent rounds, because they will not be in the *Queue* any more. These actual terminal states will be merged at earlier stage.

Another advantage of the use of *Queue* is that it is guaranteed for minimizing the overall state space at last due to an empty *Queue*. As long as the *Queue* is empty, there is no splitter-block in the initial partition, and the entire LTS is considered for minimization.

Therefore, maintaining a state list like *Queue* makes the use of on-the-fly minimization more flexible.