Chapter 5. Determine the Threshold to Trigger Minimization

In last chapter we left a question to the algorithm of on-the-fly minimization in composition. At line 18 in Figure 4.6, the condition of deciding when to start minimizing the current partially composed LTS has not been specified yet. Several approaches we had thought about are presented in this chapter. We first demonstrate a manually configurable approach to examine how to make an efficient decision.

5.1 Manual Configuration

At the beginning, we simply take the *state number* into account just because it is the easiest way to do. In this way, an argument, state number is input to the program to specify the upper bound of the enumerated state number in each round. A state counter is kept to count on enumerated states. When the number of the state counter is bigger than the specified upper bound, it is time for minimizing the current state space. After minimization, the state counter is reset to zero and continues to count on later enumerated states.

Another more accurate method is to consider the *size of objects* of both states and transitions enumerated. We manually specify the upper bound of (physical) memory

size, and keep track of the total size of enumerated state and transition objects. In case the total size of created objects is near to the specified memory upper bound, we minimize the partially composed LTS and try to gain more space for later enumeration. However, by this approach it takes time to estimate the state space after each state enumeration.

After we experiment on the two approaches described above, the enumerated state space is lowered down more obviously when the specified upper bounds are set smaller. We show an example of Two-Elevator System [36] in Table 5.1. The "**Min. Criterion**" tells the composition engine to minimize the state space while the state space size reach the given number of states. When the number is set lower, the maximum enumerated state space during composition is also decreased. We globally compose the whole system and reach 3,600 states, whereas with on-the-fly minimization the number of enumerated states can be decreased to be less than 1,500.

Two-Elevator System		
Compose w/o Min.	3600 states	7951 trans
Compose w. on-line Min.	53 states	199 trans
Min. Criterion (states)	**Max. State Num.**	**Time Spent (sec.)**
10,000	3600	2.693
1,000	1464	4.576
500	963	5.978
100	601	16.422
50	573	29.311
10	534	125.96

Table 5.1. The evaluation of Two-Elevator System.

However, without surprise, we also found that the less upper bound we set, the

more time spent on the completion time. Although we may prevent thrashing, many minimizations in composition are seen as unnecessary if there is sufficient memory for continuing to enumerate state space.

5.2 Automatic Approach

In order to both keep the benefit of reducing state space in progress and avoid unnecessary minimizations, we try to determine a better triggering point for starting minimization by detecting the memory usage automatically.

The overall basic concept is that if we have sufficient memory to enumerate the state space, there is no need to invoke minimization. And we hope to do minimization just in time before O.S. begins thrashing.

5.2.1 Page Faults Delta Detection

Our first idea is to design an extra program – a *Prober*. We use it to allocate memory spaces constantly until the disk is thrashing. The major purpose of it is to examine the upper bound of current available memory size which is sufficient for enumerating state space before triggering thrashing. And then we can use the estimated value as an input argument for triggering on-the-fly branching bisimulation..

Therefore, we have to know when the thrashing begins to appear so that the

Prober terminates. The number of page faults is the most obvious information. One can easily obtain the value of page faults by invoking the function `getrusage()`, which is included in `<sys/resource.h>` library of C. However, the pure value of page faults is useless. It is an accumulative value from starting a program. A large number of page faults do not stand for the operation of disk swapping exactly, so we change our focus on the page faults delta. One can find the estimated value of every process in the Task Manager on Windows systems. A **page faults delta** is a difference of two consecutive numbers of page faults over a period of time. We found this value growing and changing dramatically when a process actually suffered thrashing. Therefore, if the *Prober* can detect a series constantly high value of page faults delta and report the most memory space allocated, our first idea may seem to work.

Nevertheless, there are too many factors had to be decided – How large of the page faults delta should be specified? How long is the period of time? How to detect the consecutive times of page faults delta that actually growing and changing? Although one can tackle these problems technically, we finally found that the estimated value is not quite accurate each time.

However, through out these experiments, we found that O.S. is really thrashing when most of the available memory are occupied by the program itself. If no other irrelevant piece of data in the memory can be swapped out, and the program has to

constantly swap its content from memory to the disk. Therefore, we find the information of the current available memory space is a useful measure.

5.2.2 Global Free Memory Detection

Because obtaining the global available memory of the system is a way better than detecting page faults, we search for several solutions to obtain the value efficiently. A short report is given below.

(1) The first solution is from Win32API. One can obtain the current available physical memory space by using the function `GlobalMemoryStatus()`. But this is only for Windows systems, of course.

(2) The second way for Unix/Linux and Cygwin is to read the system information provided by the file `meminfo` under the virtual directory `/proc`, which is maintained automatically by the system to report its activities. The text file `meminfo` contains useful information about the memory usage. We can easily parse its content to monitor the current free physical memory.

(3) The third solution is to invoke a system level function of C – `sysconf()` declared in `<unistd.h>`. It provides useful information of one page size, the total physical pages, and available physical pages. Hence, the current available memory space is easily obtained by multiplying available physical pages with the page size.

We made a comparison with these solutions. The value of free memory space obtained by (1) and (3) are almost the same, but for (2) it is often missing one page size (4K) due to the additional operation of reading the file. When it is required to detect plenty of times, the time consuming of the (3) is the least, and then the (1), and the (2) is the most. Therefore, the (3) is the best choice for most platforms using C library.

Therefore, the *Prober* can be retired from allocating memory which is now unnecessary. The estimation code can further be integrated into our composition program to determine when to do minimization. After enumerating each state, we can estimate the value of available memory space to check whether to invoke minimization or not. Therefore, the threshold to invoke minimization is dynamic and different from time to time and machine to machine.

Another good reason to detect global free memory is that the value changes along with the system activities. It would be an ideal way, if we could keep track of the actual free memory to our program.

In order to release memory space, it is important to use the pointer type for state and transition objects. During many experiments, however, most released space is hard to figure by the approaches we described above. We even try to release the memory of containers in STL, but there is still no obvious effort. In other words, although we

actually release lots of memory of states and transitions after minimization, the free memory space detected by approaches described above cannot show them immediately. We finally found that the available physical memory is referred to those unused pages not belonging to a process. It is possible for a page which is used partially but is considered unavailable totally. The released space of a part of a page is always not counted in the free memory on many systems.

We were really upset about this situation, and tried to find out other similar adequate solutions to retrieve more accurate value of the available memory space. However, it requires more effort to achieve this goal and the methods would be different from system to system, so we made a compromise between the free memory detection and the used memory estimation.

5.2.3 Used Memory Estimation

Because the actual available memory was hard to be obtained by previous approaches, we decided to estimate the released space by ourselves.

A more feasible way is to obtain the available free memory as an upper bound before enumerating the state space. More precisely, we can obtain the value with previous approaches right before invoking the composition progress. And after enumerating each state, we estimate the used memory of current generated state and

transition objects to check whether it is almost reached the upper bound.

Because the numbers of states and transitions are under our control, it is easier to know how many memory spaces were released after minimization. Nevertheless, we also lose the flexibility of keeping track of available memory to adjust to system load dynamically. Therefore, only executing the composition program without other disturbances to affect the upper bound previously detected is highly recommended.

Below we show a suggested equation to determine the condition of minimization.

(***UsedMem*** + $\boldsymbol{MaxSize_{min}}$) ÷ ***FreeMemBound*** > 98%, where

- ***UsedMem***: the memory space used by current generated state and transition objects.
- $\boldsymbol{MaxSize_{min}}$: the maximal memory space required for later minimization.
- ***FreeMemBound***: the upper bound of free memory space detected previously.

This is the proportion of used and required space to free memory. It represents the percentage of the used and required memory space. The high percentage means that the current state space occupies plenty of memory space. We preserve 2% to adjust possible inaccuracy of the upper bound and avoid fully occupying the whole memory space leading to O.S. thrashing.

The value of $\boldsymbol{MaxSize_{min}}$ is an estimating size preserved for doing minimization. We present its possible evaluation below.

$$\boldsymbol{MaxSize_{min}} = |S'|\times\textbf{\textit{sizeof}}\textbf{(Block)} + \textbf{\textit{sizeof}}\textbf{(}\text{ptr}\textbf{)}\times(2\times|S'| + |\rightarrow'|)$$

$|S'|$ denotes the state number of the partially composed LTS, while $|\rightarrow'|$ denotes the transition number. ***sizeof*(Block)** is the size of a block structure, and ***sizeof*(**ptr**)** is the size of a pointer type (generally 4 bytes). We presume that the maximum space used by the refinement algorithm is resulted in splitting all states into individual blocks. In other words, the number of states and blocks are the same. Therefore, the statement, $|S'|\times$***sizeof*(Block)**, yields the size of all block objects probably created, and the latter, ***sizeof*(**ptr**)**$\times(2\times|S'| + |\rightarrow'|)$, represents the size of pointers to objects of blocks held in block lists, and objects of states and transitions held in bottom lists, non-bottom lists, and transition lists respectively.

The next is the estimating equation of ***UsedMem***.

$$\boldsymbol{UsedMem} = |S'|\times\textbf{\textit{sizeof}}\textbf{(State)} + |\rightarrow'|\times\textbf{\textit{sizeof}}\textbf{(Transition)} + 4\times|\rightarrow'|\times\textbf{\textit{sizeof}}\textbf{(}\text{ptr}\textbf{)}$$

The statement, $|S'|\times$***sizeof*(State)** + $|\rightarrow'|\times$***sizeof*(Transition)**, calculates the size of current generated state and transition objects. The last, $4\times|\rightarrow'|\times$***sizeof*(**ptr**)**, calculates those transition pointers may held by each state. According to the structure of a **State**, each transition must be held by its source and target states, so a non-τ transition is referenced twice and a τ transition is even referenced double twice. We multiply 4 times for each transition pointer for saving efforts to identify whether it is a

τ or non-τ transition.

Although overriding the operator `new` and `delete` and further taking account of extra implicit space used by STL containers may estimate the values more accurately than above, the time spent on computing will increase substantially after each state enumeration. It is difficult to obtain a balance between time and space.

At last, we have to notice that these approaches described are not total solutions to all situations. We merely propose a possible way to alleviate the occurrence of bad conditions gradually. If a quite large subsystem occupies most available space after minimization, it is still hard to prevent the O.S. from thrashing. This is the problem of state explosion which make minimization no longer feasible.