Weryfikacja wspomagana komputerowo

2016-2017

Lab 4 – Leader Election and memory usage

Vincent Penelle

Topics of this Lab

- A modelisation of a really concurrent program, this time.
- Exploration of the memory options of SPIN.

Exercise 1: Leader Election (understanding the problem) We consider a set of N processes, each having a unique integer as identifier. They can communicate only in a directed ring, i.e. the first can send a message to the second, the second to the third, etc... They are trying to determine which one has the biggest identifier (they don't know how many they are, and they don't know which number are present).

For simplicity reason, you will only consider cases in which the identifiers are $1, \cdots, N$, but the algorithm should work for any set of disctinct identifiers.

1. Give an algorithm for each process that solve that problem (do it on paper and show me). You will model the communication between two process by a channel.
2. What is the maximum number of messages exhanged by this algorithm?

Exercise 2: A naive implementation Consider this implementation[1] of the leader election algorithm (hopefully, it is similar to what you got), test and understand how it works.

1. Write a LTL formula checking that a leader will eventually be elected (and test it).
2. Write a LTL formula checking that the process with the biggest identifier will eventually be selected (use labels and use the fact that you know which process is the maximum).
3. Write a LTL formula checking that the number of leader is 0 until it becomes 1, and after that it stays 1.

Exercise 3: Memory Usage Test the previous claims with and without the options in the Storage mode tab, and the Search mode tab. Observe the differences in terms of number of states explored and memory usage.

[1] `./leader-tres-naif.pml`

Technical point:

We are now touching one of the limitation of Spin: if a model is too big, your computer may not be able to run a exhaustive verification without dying (figuratively) in the process. To do so, there are some verification options which can be taken to reduce the size of the model or the memory it takes. You can find some documentation on that on the spin documentation, e.g. this one[a].

Here is a quick summary of what you should at least have in mind:

- *exhaustive storage:* the automaton will be fully constructed and stored. Exhaustive verification, but costly in memory.
- *minimized automaton:* the automaton will be minimized. The memory cost will be slightly less, at a cost of a bigger time calculus, as the minimization algorithm is exponential (at worst).
- *collapse compression:* Spin will compress the information it stores to reduce slightly its memory usage.
- *hash-compact:* By storing similar sub-automata into an hash table, Spin will reduce the size of the overall automaton. This may lead to some incompleteness in the search (you may not see undesirable behaviour).
- *bitstate/supertrace:* Reduces a lot the size of state storage cost compared to the hash-compact. The exhaustive coverage is not guaranteed a lot, but it is usually successful to identify correctness violation (be warned: usually is not always).

Apart from tricking memory, you can as well prevent the exploration of a lot of branches by forcing execution strategies. In the search mode, you have:

- *partial order reduction:* this option merges paths differing only by the interversion of some instructions without changing the overall state of the program. This reduces a lot the number of states of the model, while being precise for most checks. Keep in mind that if you are interested in checking properties on the trace of the program rather than the configurations, you may want to not use it.
- *bounded context-switching:* Spin will stick to executions which does not change of the process which is executing a step more than a bound you fixed (warning: it is something like that, but probably a bit more complicated). That will reduced the space search. It will not be exhaustive, but may be sufficient to find counter-example to safety properties (if a state is reachable in less than N context-switches, it is reachable).
- *iterative search for short trail:* This will try to search the shortest error trail. That may take more time and memory than without it, but you'll have a more readable error trail.
- *breadth/depth-first search:* in case you are running on a several core machine, spin will split executions to favorise a depth or a breadth search. It may lead

to different times before reaching the error (a priori, that is all - apart that you don't have access to the same options).

Anyway, even if you can use these tools, keep in mind that the best way to verify a model in a reasonable time and memory is too have a simple model in the fist place, so try to minimise the complexity while coding your model first.

[a] `http://spinroot.com/spin/Man/4_SpinVerification.html`

Exercise 4: Another leader election algorithm In the previous case, we had a big number of maximum message exchanged. We will try to lower that number.

The idea is that now, each process will have two states: active and passive. At the beginning, everybody is active.

If a process is active, he takes two messages of its two predecessors (p_1 then p_2) and compare them with his number n. If p_1 is bigger than the two others, it updates its number to p_1 and stays active. Otherwise, it becomes passive.

If a process is passive, he just passes down whatever he receives.

If an active process receives a message equal to its number as its first message, he concludes that he is the only left active process, send a special message of victory, and then every process passes it down, determining on the way if he is the leader or not (by comparing with its initial number), and when the victory message reaches the sender, he shuts down.

1. Write an algorithm (or a drawing, or both) explaining this algorithm.
2. How many messages are passed at worst? At best?
3. Here[2] is an implementation of this algorithm. Test it.
4. Verify the previous claims (and compare with the results of complexity of it).
5. Verify with a LTL property than every process always receives a alternation of messages with one and two, until it receives a victory message.
6. What else can you check?

[2] `./leader.pml`