|  |
| --- |
| Alejandro Cirugeda K-5087  Juan Carlos Gomez K-5101 |

|  |
| --- |
| Automatic program verifier (partial exploration, with compression) |

# Summary

The objective is to create a program to conveniently investigate the unfolding behavior of a protocol and discovers if it presents some faults. In order to explore the full protocol, we build a reachability state space checking each global state and see if present some faults such as deadlocks.

In order not to run out of memory we will implement a compression mechanism to reduce the amount of memory needed to run the program. We will use a bistate hashing, which will convert a global state into a numerical number which points to a bit in a computer memory, this bit will indicate if the node has been visited.

## FSM

The protocol will be represented as a set finite state machines, each one with its own number of states, transition, The FSM will be read from a simple file in a txt format, using simple software such as notepad++.

Ex:

M0:S0,S1

t:S0+aS1

t:S1+bS0

t:S0-aS1

Where the first two characters (M followed by a number) indicates that an FSM has to be created with its id. The different transitions are represented by a “t” followed by an upper case ‘S’ (which represents the initial state), sending (+) or receiving (-) a signal in lower case and the next state with capital letter. Every transition is in a different line of the document.

With this format we will know all the necessary information for execute the verifier.

While the file is being read, we create two classes: the ‘FiniteMachine’ class, which stores all the FSM’s functionality, a ‘State’ class, which represents the different states of the FSM and a ‘Transition’ class, which stores all the information about the FSM’s transitions.

The way in which we are going to create this classes is the following: every time we read an M, we create a ‘FiniteMachine’ class. In the same way, when we read an upper case ‘S’, we create a ‘State’ class, which is going to store all the information about the transition, which is also created when the program reads a lower case ‘t’. The information about the transition is saved in the origin state, and this one is also stored in the FSM. Finally, all the created FSMs are saved in a list.

These three classes have this information:

1. FiniteMachine:

* Id: The identifier of the FSM.
* States: The list in where we store the FSM’s states.
* Actual\_state: Indicates the initial state.

1. State:

* Id: The identifier of the state.
* Transitions: The list in where we store the transitions which has this state as its original one.
* Counter: It counts how many states we visited.

1. Transition:
   * Id\_fsm: The identifier of the FSM which has this transition.
   * Actual\_state: The state of origin.
   * Next\_state: The state of destination.
   * Action: It indicates if the signal is being sent or received.
   * Signal: It indicates the information which is sent or received.

Also, for the execution of the algorithm we use the “Node” class which stores information about the current state and the possible transitions to other nodes.

1. Node:
   * Global\_state: It stores the global state of the transition.
   * Transaction\_list: A list of possible transitions from this node.

## Hashing function

The hashing function will oversee converting the global state into a numerical value. This numerical value will be 16bits unsigned integer in order to point a position in the file. We took inspiration from “one\_at\_a\_time” Jenking’s hashing function and we tried to create our own version modifying the number of positions which the bits shift.

The value will be used to point the corresponding bit which represents the global state that we are looking for.

In order to save Ram, we will store all the necessary bits in a file. This operation will be slower that using RAM memory, but with this method we won’t run out of memory during the execution of the verifier.

## Algorithm

Once we store the FSM we can start with the verification of the program. We start in the initial state with all the channels empty and start checking all the following reachable global states in order to find any possible faults such as deadlocks.

We use depth-first algorithm in order to go through the reachability graph. Every state we visit we check with the bit-state-hashing function if the state has been already visited. If it’s a new state, we check for the if the Global State have a deadlock with this function:

Put code check deadlock and explain

After checking if has deadlock we introduce the possible child in the stack and continue the execution. We limited the stack in order to avoid a problem with unboundedness and an infinite execution.

Eventually the execution will finish once the stack is empty. Some states will be ‘hidden’ due to the collision with the hashing function.

1. Experiments

Imagen que contiene texto

Descripción generada automáticamenteFirst, we prove our programme with the following FSMs:

And we have the next output:

Imagen que contiene texto

Descripción generada automáticamente

As we can see the programm stores both FSM, with their states and transitions. In the output it is also shown the amount of deadlocks.

Imagen que contiene texto, pizarra

Descripción generada automáticamenteThe second FSM to be tested are the following:

Resulting in the next output:

Imagen que contiene interior, naranja

Descripción generada automáticamenteImagen que contiene sentado, electrónica

Descripción generada automáticamente