Model Checking for Programming Languages using VeriSoft

by Patrice Godefroid

Overview

- First paper to introduce the idea of Model Checking for Programming Languages, in 1997
- Existing Model Checking techniques verify properties of models (abstractions), not the actual implemented systems
- Extends model checking to descriptions of concurrent systems written in full-fledged languages like C/C++
- Introduces a new search algorithm for checking such systems
- Implements the algorithm in VeriSoft: a framework that explores the state-space of systems composed of several concurrent processes executing arbitrary C code

What is Model Checking

- Model Checking = verification by systematic state-space exploration = exhaustive testing
- Model Checking = "check whether the system satisfies a temporal-logic formula"
- Simple yet effective technique for finding bugs in high-level hardware and software designs (e.g., FormalCheck for HW, SPIN for SW)
- Used to analyze the correctness of concurrent systems and check properties like deadlocks, assertion violations, dead code, etc.
- Primary goal: Find errors that would be hard to detect and reproduce otherwise, rather than mathematically proving their absence!

Concurrent Systems and Dynamic Semantics

Assumptions

- P: a finite set of processes
- O: a finite set of communication objects
- Processes communicate through objects defined as a pair (V, OP):
  - V = domain (possible values)
  - OP = operations that can be performed
  - Examples: shared variables, semaphores, FIFO buffers
  - Operations on a specific shared object must be mutually exclusive
- Visible operations: Operations on communication objects (!= invisible local operations)
- Blocking operation: An operation that cannot be completed
- At any given time, the concurrent system is in a state
- global state: next operation executed by every process in the system is visible (s0: first and unique global state - initial global state)
- (process) transition: visible operation followed be a finite amount of invisible ones, performed by a single process

Concurrent Systems and Dynamic Semantics

Assumptions

- T: set of all transitions in a system
- disabled (transition): it's global operation is blocking (!= enabled)
- all enabled transitions in a state s can be executed by it
- number of invisible operations are finite -> execution of an enabled transition always terminates and we arrive to a new global state s' (successor of s by transition t).
- s' can be reachable by s through the execution of a finite sequence of transionts w
- Concurrent system == closed system: from it's initial global state, it can evolve and change its state by executing enabled transitions
- We restrict AG to the global states and transitions that are reachable from s0 - the other ones play no role in the behavior of the system

Example

Some ancient C

```c
/* phil.c : dining philosophers (version
without loops) */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define N 2

philosopher(i)
  int i;
{
  printf("philosopher %d thinks\n",i);
  semwait(i);        /* take left fork */
  semwait((i+1)%N);  /* take right fork */
  printf("philosopher %d eats\n",i);
  semsignal(i);      /* release left fork */
  semsignal((i+1)%N);/* release right fork */
  exit(0);
}
```

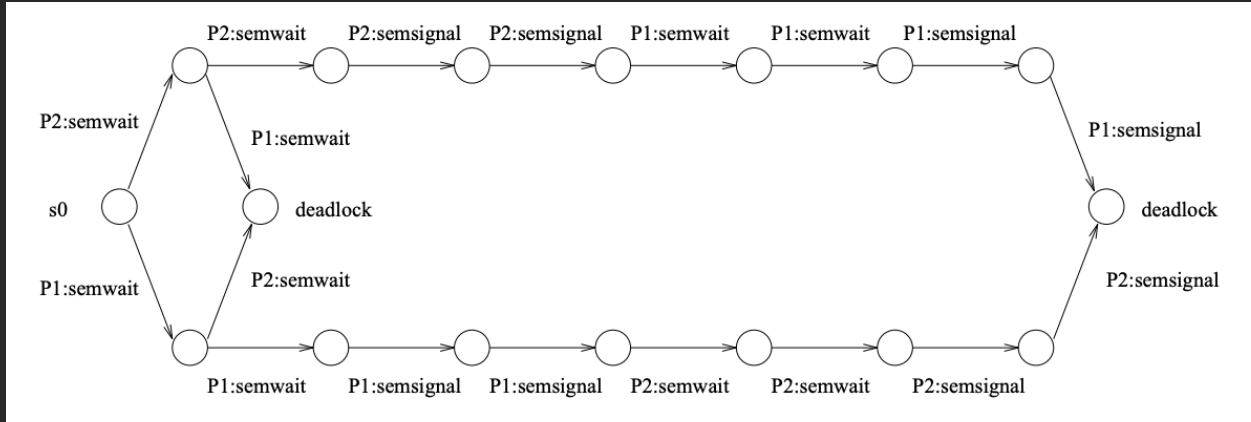
```c
main()
{
  int semid, i, pid;
  semid = semget(IPC_PRIVATE,N,0600);

  for(i=0;i<N;i++)
    semsetval(i,1);

  for(i=0;i<(N-1);i++) {
    if((pid=fork()) == 0)
      philosopher(i);
    };

  philosopher(i);
}
```

Example



Properties to check

What are we looking for?

- Deadlocks -> next operation for all processes is blocking
- Detection of violation of assertions -> added by user with a VeriSoft function

Theorem 1

Consider a concurrent system as defined above, and let AG denote its state space. Then, all the deadlocks that are reachable after the initialization of the system are global states, and are therefore in AG. Moreover, if there exists a state reachable after the initialization of the system where an assertion is violated, then there exists a global state in AG where the same assertion is violated.

Or Simply...

Deadlocks and assertion violations can be detected by exploring only the global states of a concurrent system.

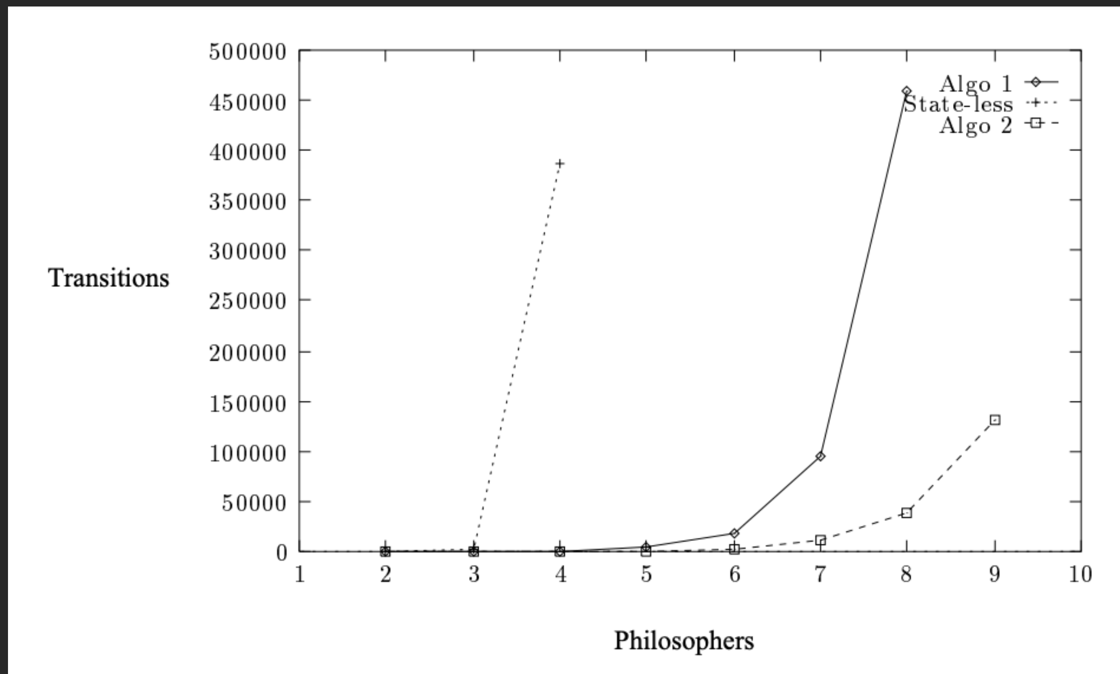
Proof

Not here luckily, refer to Appendix

Existing State-Space Exploration Techniques

- Create AG using a basic search algorithm (BFS/DFS)
- This algorithm assumes that each state can be represented with a unique id
- Programms written in full fledged programming languages need a lot of data to determine a unique id -> inefficient
- We can do state-less exploration: systematically searching without storing intermediate states
- Visits a lot more states, very slow compared, might not terminate

Existing State-Space Exploration Techniques



Classical: 708 transitions Stateless: 386816

An Efficient Stateless Search Algorithm

An introduction to partial-order methods

- Algorithms that were developed in order to tackle the "state explosion" problem
- Are used to check properites of the state space without having to construct all of AG :
  - If the same set of events are executed by to processes that don't interefere with each other, then these executions are independent.
  - Changing their order doesn't affect anything

Definition 1

Definition 1 Let $\mathcal{T}$ be the set of system transitions and $D \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. The relation D is a *valid dependency relation* for the system iff for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \notin D$ (t_1 and t_2 are *independent*) implies that the two following properties hold for all global states s in the global state space A_G of the system:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other); and
2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

Definition 1 takeaways

1. Independent transitions can neither enable or disable each other
2. We have commutativity on independent transitions.

An Efficient Stateless Search Algorithm

Defining (in)dependency in a concurrent system is non-trivial but crucial (refer to other paper for more information)

Selective-Search

Compute a much stricter set T of enabled trsanstions in a state s and explore only them.

- Techniques to compute such sets T are persistent set and sleep set algorithms. Ts are called persistent sets
- Each transition in a persistent set in s (T) (and the states they lead to) is independent to each transition (and the states the they lead to) that is not in T

Persistent-Sets

Partial-order method that makes use of the static structure of the system. They are their category of algorithms, each differe just by how they make use of the information.

Sleep-Sets

Partial-order dynamic pruning method that has shown empirically that most states are visited once, without having incomplete verification and not having to store many of the states. (Definition 2)

Combine the two of the above...

You get and efficient state-less search algorithm

An Efficient Stateless Search Algorithm

Algorithm 2

```
1   Initialize: Stack is empty;
2   Search() {
3     DFS(∅);
4     }
5   DFS(set: Sleep) {
6      T = Persistent_Set()\Sleep;
7      while T ≠ ∅ do {
8         take t out of T;
9         push (t) onto Stack;
10        Execute(t);
11        DFS({t' ∈ Sleep | t' and t are independent});
12        pop t from Stack;
13        Undo(t);
14        Sleep = Sleep ∪ {t};
15        };
16     }
```

selective DFS search in state space of concurrent system

- Stack: contains the sequence of transtions that lead up to the current global state s (how we got here)
- Sleep (set): each global state (in reallity set of transitions) reached during the search (what to skip)

An Efficient Stateless Search Algorithm

Theorem 2

Consider a concurrent system as defined in Section 2, and let AG denote its state space. Assume AG is finite and acyclic. Then, all the dead locks in AG are visited by Algorithm 2. Moreover, if there exists a global state in AG where an assertion is violated, then there exists a global state visited by Algorithm 2 where the same assertion is violated.

Or simply...

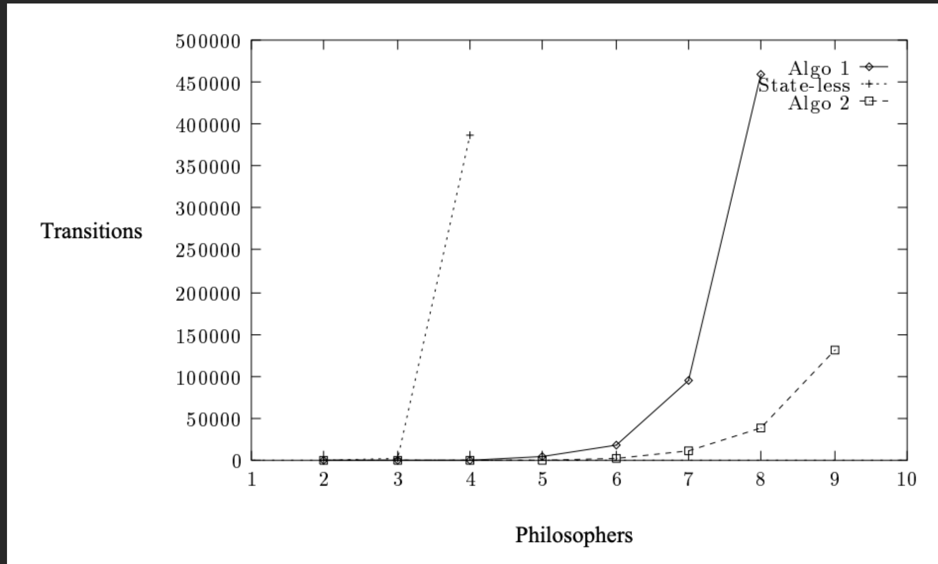
With the search described in Algorithm 2 it is guaranteed that deadlocks and assertion violations will always be found

Proof

Also not here, refer to Appendix

An Efficient Stateless Search Algorithm

- Algorithm 2 is granted to terminate only when it is acyclic and finite
- Either way, very efficient for exploring the state-space of any concurrent system
- Does not store any state in memory and explores less states than a classical search (Algorithm 1)



VeriSoft

- Tool for systematically exploring the state space of systems (concurrent C processes)
- Has implemented Algorithm 2, with limited depth
- Every process is mapped to one UNIX process
- Execution of those managed be a "scheduler" process: observes the visible operations and can suspend their execution
- Deadlock/assertion violation detected -> execution stopped and transitions in Stack are exhibited to the user
- Interactive graphical simulator/debugger also available for following the execution of the processes.
- Can also catch livelocks and divergencies

! A state-less search cannot detect cycles and is thus restricted to the verification of the safety properties above.

VeriSoft

VeriSoft Simulator – Trace View

File | Reset | Next Event | Move | Go To End | Quit

Process_1 Process_2

VS_toss(3)=1

send_to_queue(1,10,room_is_hot)

rcv_from_queue(1,10)=room_is_hot

VeriSoft Simulator – (Pruned) State Space View

Home | Zoom In | Zoom Out | Labels | Quit | buttons: L=go to state; M=center view; R=examine

Move the pointer over a node to see which state it represents.

Key: Assertion Violations: 1 Deadlocks: 0 Aborts: 0

VeriSoft Simulator

Assertion violation!

Dismiss

VeriSoft Simulator – Process 2

Step | Next | Continue | Print | Quit

VeriSoft Simulator – Process 1

Step | Next | Continue | Print | Quit

```
    is_door_closed=1;
    if (is_room_hot)
      ac=1;
  };

  /* test */
  if (is_room_hot && is_door_closed)
    VS_assert(ac);

  };
}
void Environme
{
```

VeriSoft Simulator – State Space Filter

Text Regular Expression:

Match | Clear | Quit

AND

Labels

send_to_queue(1,10,room_is_hot)
send_to_queue(1,10,room_is_cool)
send_to_queue(1,10,open_door)
send_to_queue(1,10,close_door)
rcv_from_queue(1,10)=room_is_hot
rcv_from_queue(1,10)=room_is_coo

Processes

1
2

pts/2 /home/god/verisoft/examples/ac-controller

```
comeback $ verisoft main.c -simul error1.path
gcc -I/home/god/verisoft/bin /home/god/verisoft/bin/verisoft_simul_SunOS_5.5.1.o -
DVERIFY -g main.c
/home/god/verisoft/bin/simul.tcl error1.path
Loading sss.VS for state space view (please wait)...
Done.
```

VeriSoft: Example Use

VeriSoft successfully discovered an error in a 2500-line concurrent C program controlling robots operating in an unpredictable enviroment:

- Six processes that communicate via shared memory and semaphores.
- Two of the processes control robots that collect objects randomly dropped on a table by a third robot, represented by a third process.
- The three other processes are used to simulate the rest of the environment of the robots.

Sometimes the two robots that collect objects on the table suddendly stop moving. A problem like this would be hard to reproduce. After some minutes, VeriSoft reported a scenario composed of 29 transitions that led to divergence: an error in a `while` loop in the code of one of the provesses. This blocked other processes too.

Overall

- Model Checking can be used to check desirable properties for concurrent systems
- Provides an efficient state-less search algorithm and implements it in VeriSoft.
- Is a powerful technique for system testing and verification.
- Has been cited by many important works like KLEE, DART, S2E.
- Is still relevant to this day.

Thank you. Any questions?