Model Checking Programs

http://d3s.mff.cuni.cz

Pavel Parízek

FACULTY OF MATHEMATICS AND PHYSICS
Charles University
Model checking

Structure M

Formula f

LTL: \( p \Rightarrow F q \)

Verification task: \( M, s \models f \) ?
Model checking SW and HW

• Goals
  ▪ Systematic exploration of all possible behaviors
    • Example: all possible interleavings of concurrent threads
  ▪ Checking required properties in each state (path)

• Model
  ▪ Source code (binary) ➔ program state space

• Property
  ▪ assertion, deadlock freedom, no data races, ...
Program state space

- Directed graph
  - States
  - Transitions
Q: What does a program state contain?
States

- Local state of each thread
  - Program counter (PC)
  - Call stack (parameters, local variables, operands)

- Global state shared between multiple threads
  - Heap objects (field values) and pointers
  - Status of each thread (Runnable, waiting, ...)
  - Thread synchronization primitives (locks)
Q: What about transitions?
Transitions

- Statements (instructions)
  - Updating states (PC, variables)
Program state space

- Directed graph
  - States
  - Transitions
  - what else?
Program state space

- Directed graph
  - States
  - Transitions
  - Choices
Q: What types of choices there are?
Choices

- Thread scheduling

- Data
  - Unknown inputs
Program state space

- States
- Transitions
- Choices

PC: 3, i: 0
PC = PC+1, i++

PC: 4, i: 1
PC = PC+1, i = choose-int(2,3)

PC: 5, i: 2
PC: 5, i: 3
Example: producer – consumer

```java
public Producer extends Thread {
    void run() {
        while (true) {
            buf.add(++i);
        }
    }
}

public Consumer extends Thread {
    void run() {
        while (true) {
            i = buf.get(0);
            print(i);
        }
    }
}

public static List buf;

(new Producer(var)).start();
(new Consumer(var)).start();
```
Terminology

- **Reachable state space**
  - From the initial program state

- **Error state**
  - $E$

- **Safety**
  - Error state is not reachable
Properties

• Categories
  - State
  - Path
Q: Divide properties into categories

Properties
no deadlock
data race
assertion
LTL formula

Category
state
path
<table>
<thead>
<tr>
<th>Properties</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>no deadlock</td>
<td>state</td>
</tr>
<tr>
<td>assertion</td>
<td>path</td>
</tr>
<tr>
<td>LTL formula</td>
<td>multiple paths</td>
</tr>
<tr>
<td>data race</td>
<td></td>
</tr>
</tbody>
</table>
State space traversal
State space traversal

- Explicit traversal of the concrete state space
- SAT-based traversal of symbolic state space
Explicit state space traversal

- DFS: depth-first search
  - From the node corresponding to the initial state

- Properties checked in each state
  - Error state reached $\Rightarrow$ counterexample

- Counterexample (error trace)
  - Path in the state space that violates given property
Explicit state space traversal with DFS

INIT
  visited := \{s0\}
  push(stack, s0)
  DFS(s0)
end INIT

DFS(s)
  for each t in enabled(s) do
    s' := t(s)
    if not P(s') then
      counterexample := stack
      exit
    if s' not in visited then
      visited := visited + \{s'\}
      push(stack, s')
      DFS(s')
      pop(stack)
  end for
end DFS()
Explicit state space traversal with DFS

INIT

\[
\text{visited} : = \{s_0\}
\]

push(\text{stack}, s_0)

DFS(s_0)

end INIT

DFS(s)

for each \( t \) in enabled(s) do

\[
s' := t(s)
\]

if not P(s') then

counterexample := stack
exit

if \( s' \) not in \text{visited} then

\[
\text{visited} := \text{visited} + \{s'\}
\]

push(\text{stack}, s')

DFS(s')

pop(\text{stack})

end for

end DFS()
Explicit state space traversal with DFS

**INIT**

- visited := \{s0\}
- push(stack, s0)
- DFS(s0)

**end INIT**

**DFS(s)**

- for each \( t \) in \( \text{enabled}(s) \) do
  - \( s' := t(s) \)
  
  - if not \( P(s') \) then
    - counterexample := stack
    - exit
  
  - if \( s' \) not in visited then
    - visited := visited + \{s'\}
    - push(stack, s')
    - DFS(s')
  
- pop(stack)

**end for**

**end DFS()**

Executing transitions
Explicit state space traversal with DFS

```
INIT
    visited := {s0}
    push(stack, s0)
    DFS(s0)
end INIT

DFS(s)
    for each t in enabled(s) do
        s' := t(s)
        if not P(s') then
            counterexample := stack
            exit
        end if
        if s' not in visited then
            visited := visited + {s'}
            push(stack, s')
            DFS(s')
            pop(stack)
        end if
    end for
end DFS()
```

Evaluating properties
Explicit state space traversal with DFS

INIT
visited := \{s0\}
push(stack, s0)
DFS(s0)
end INIT

DFS(s)
for each \(t\) in enabled(s) do
\(s' := t(s)\)
if not \(P(s')\) then
    counterexample := stack
    exit
if \(s'\) not in visited then
    visited := visited + \{s'\}
push(stack, s')
    DFS(s')
    pop(stack)
end for
end DFS()
State space traversal with DFS – example

Random rnd = new Random();
int i = 2;
int j = 0;

int c = rnd.nextInt(3);

if (c == 1)
    j++;
else if (c == 2) {
    j = 1;
    c = 1;
}

int k = i / j;

Stack: 1,2,6
Visited states: {1,2,3,4,5,6}
Random rnd = new Random();
int i = 2;
int j = 0;

int c = rnd.nextInt(3);

if (c == 1)
    j++;
else if (c == 2) {
    j = 1;
    c = 1;
}

int k = i / j;

Stack: 1,2,7
Visited states: {1,2,3,4,5,6,7}

1 init
2 i = 2, j = 0
3 i = 2, j = 0, c = 1
4 i = 2, j = 1, c = 1
5 i = 2, j = 1, c = 1, k = 1
6 i = 2, j = 0, c = 2
7 i = 2, j = 0, c = 0
8 division by zero!
Model checking programs: limitations
Limitations

- Decidability
  - For many interesting programs and interesting properties, model checking is undecidable
  - Example: assertion checking
    - Undecidable for multi-threaded programs with procedures
    - Decidable for single-threaded boolean programs
Limitations

- Possibly infinite state systems

Q: What can make the state space infinite?
Limitations

- Possibly infinite state systems
  - Data types with large or infinite domains (int, float)
  - Unbounded heap and number of threads
  - Unbounded recursion of procedure calls (stack)

- Remedy: abstraction
Limitations

- State explosion
  - a non-trivial program has too many states
  - the state space contains too many choices

- State space size exponential with respect to
  - Number of threads
  - Size of data domains
State explosion

- High number of concurrent program threads
- Many instructions executed by each thread

\[
M = \frac{(\sum_{i=1}^{N} n_i)!}{\prod_{i=1}^{N} (n_i !)}
\]
State explosion

- Consequences
  - Exploring too many choices, states, and transitions
  - Storing too many states in memory
  - model checker runs out of memory and time

- Model checking of large and complex programs is not practically feasible
  - ... but many research teams are working on this
Q: So what can we do with state explosion?

T1: a ; b
T2: c ; d
Partial order reduction

- Most transitions perform operations local to a given thread
  - Examples: arithmetic over stack operands (in Java), updating local variables

- Global operations (statements)
  - Field access on a shared heap object
  - Thread synchronization (lock, wait)
Partial order reduction

- Independent transitions
  - Performing only thread-local statements
  - All their interleavings give the same result
Partial order reduction

- Independent transitions
  - Commutative $\rightarrow$ any ordering is valid
  - Execution of one does not disable others

- All the possible interleavings of independent transitions from a given state are equivalent
Partial order reduction

• Practical approach
  ▪ Scheduling choices only at statements that represent communication among threads (conflicts)

• Communication statement
  ▪ may have effects visible to other concurrent threads
  ▪ may depend on other threads by reading shared data

• Why thread choice
  ▪ Let other threads react or modify shared data
Addressing state explosion

- Symmetry reductions
- Heuristics
Symmetry reductions

- Two states: $s1, s2$
  - State matching: $s1 \neq s2$
  - Program execution: $s1 = s2$

- Goal: avoid repeated processing of such states

- Approach
  - Divide state space into equivalence classes
  - Explore only **canonical representation**
Symmetry reductions

- Class loading order
- Heap addresses

- Partial order reduction
Class loading symmetry

• Program execution
  - Actual position of class data in the static area does not influence observable behavior

• Model checkers
  - Internal representation of program states
  - Class loading order matters in some cases

• Solution
  - Canonical representation of the static area
    • Fixed order of class loading over all state space paths
Heap symmetry

• Program execution
  ▪ Exact address of a heap object does not influence observable behavior

• Model checkers
  ▪ Internal representation of program states
  ▪ Heap shape and layout matters in some cases

• Solution: heap canonicalization
  ▪ Canonical addresses of heap objects
  ▪ Issues: garbage collection, deallocation
Heuristics

- **Motto**
  - “find an error before the model checker runs out of memory and time (resources)”
  - Better testing: find many errors in reasonable time

- **Approach**
  - Focus on state space fragments with errors
    - Guide model checker towards possible error states
    - Identify and drop error-free parts of the state space
“standard” DFS

INIT
visited := \{s0\}
push(stack, s0)
DFS(s0)
end INIT

DFS(s)

\textbf{workSet} := \text{enabled}(s)
for each t in \text{workSet} do
\hspace{1em} s' := t(s)
\hspace{1em} if not P(s') then
\hspace{2em} counterexample := stack
\hspace{2em} exit
\hspace{1em} if s' not in visited then
\hspace{2em} visited := visited + \{s'\}
push(stack, s')
DFS(s')
pop(stack)
end for
end DFS()

BeFS + heuristics

INIT
visited := \{s0\}
push(stack, s0)
BeFS(s0)
end INIT

BeFS(s)

\textbf{workList} := \text{order}(\text{enabled}(s), h)
for each t in \text{workList} do
\hspace{1em} s' := t(s)
\hspace{1em} if not P(s') then
\hspace{2em} counterexample := stack
\hspace{2em} exit
\hspace{1em} if s' not in visited then
\hspace{2em} visited := visited + \{s'\}
push(stack, s')
BeFS(s')
pop(stack)
end for
end BeFS()
Heuristic functions

- Random walk (search)
- Branch coverage
  - Preferring unexplored paths at branching point
- Maximize thread switching
- Prioritize selected threads
- Prefer most blocked threads
- ... and many others
Heuristics functions

- Problem: may not give the best/correct answer
  - Error states usually identified on-the-fly during state space traversal

- Consequences
  - Dropped state space fragments with errors inside
  - Misguided search towards error-free state space

Success not guaranteed !!
Practical issues

- Relaxed memory models (e.g., JMM for Java)
- Mapping counterexamples to source code
- Efficient management of program states
  - Operations: storage, state matching, backtracking
  - Transitions modify a small part of program state
    - Keep only “diffs” from the previous state on the path
  - Comparing hash values ➞ possible collisions
Further reading


