

# Bounded Model Checking and Feature Omission Diversity

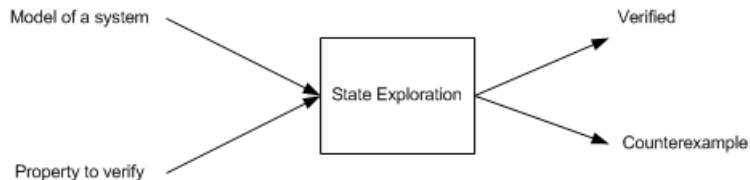
Amin Alipour and Alex Groce

Oregon State University

November 10, 2011

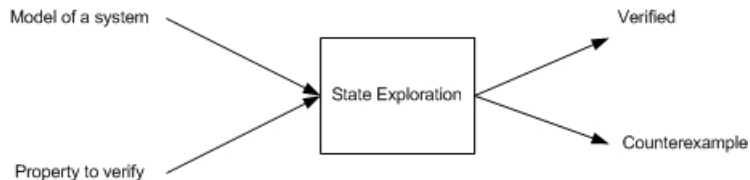
# Model Checking

Model checking is a formal verification technique which inspects all reachable states for violations of a specification.



# Model Checking

Model checking is a formal verification technique which inspects all reachable states for violations of a specification.



- The number of states is *exponential* in number of variables.
- The number of states is *exponential* in number of threads.

**State-space explosion problem!**

# Remedies for state-space explosion problem

- *Symbolic model checking* uses binary decision diagrams to represent the transition system.
- *Abstraction* simplifies the model.
- *Partial order reduction* identifies the independent interleaving of threads.

# Bounded Model Checking

In Bounded Model Checking (BMC) , the program is unrolled for a given length  $k$  and it is reduced to a SAT problem, s.t. the solution to the SAT problem is a counter-example for the program.

- Example: if  $R$  describes the relation between the current state and the next state and a safety property  $P$ , the SAT problem looks like:

$$\begin{aligned} & I(S_0) \wedge \\ & (R(S_0, S_1) \wedge R(S_1, S_2) \wedge \dots \wedge R(S_{k-1}, S_k)) \wedge \\ & (\neg P(S_0) \vee \neg P(S_1) \vee \dots \vee \neg P(S_k)) \end{aligned}$$

# The problem is still there!

Even with the power of symbolic techniques, the state-space explosion is a fundamental problem for model checking:

- real-world systems have very large state spaces, and
- effective abstractions and symbolic representations are challenging to discover.

# Realities to face

- There is usually limited time budget for model checking.
- In large systems, model checking techniques use lot of time and memory.

# Realities to face

- There is usually limited time budget for model checking.
- In large systems, model checking techniques use lot of time and memory.

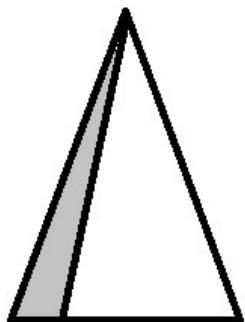
**How to make them faster?**



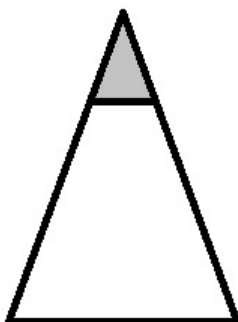
# Swarm Verification

- Swarm verification exploits the multi-core processors to make exploration of new states faster by diversification of search strategies.
- *Key idea:* When no heuristic is known to speed of the search, try several diverse search strategies in parallel.

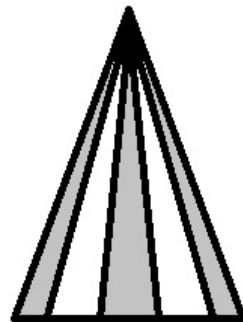
# Diversification of Search Strategies



Depth First  
Search



Breadth  
First Search



Randomized  
Search

# Features

Consider the following components in a program:

- functions,
- statements, or
- particular range of valuations for variables.

# Features

Consider the following components in a program:

- functions,
- statements, or
- particular range of valuations for variables.

**Typically, a counterexample is made by some of them, not all of them!**

# Swarm Testing

Swarm testing generalizes the search diversification of swarm verification beyond the choice of a depth-first search strategy to the selection of *test features*.

- A test feature is a predicate over test cases, controlled by the test generation process.
  - *Example:* In test of a file system, a test feature might be whether the test case includes calls to `close`.
- Swarm testing tries several test sessions with different test features in parallel.
- Swarm testing achieves better fault detection and code coverage.

# Swarm bounded model checking

Is there a way for swarm bounded model checking?

## Example – stack

Assume log statements represent a feature.

```
#define SIZE 64
int s = 0;
int stack[SIZE];
int top() {
    log("top");
    return stack[s];
}
void push(int i) {
    log("push");
    stack[s++] = i;
}
void pop() {
    log("pop");
    if (s > 0) {
        s--;
    }
}
```

## Example – stack

Assume log statements represent a feature.

Errors:

- Array out of bounds.

```
#define SIZE 64
int s = 0;
int stack[SIZE];
int top() {
    log("top");
    return stack[s];
}
void push(int i) {
    log("push");
    stack[s++] = i;
}
void pop() {
    log("pop");
    if (s > 0) {
        s--;
    }
}
```



## Example – stack

Assume log statements represent a feature.

```
#define SIZE 64
int s = 0;
int stack[SIZE];
int top() {
    log("top");
    return stack[s];
}
void push(int i) {
    log("push");
    stack[s++] = i;
}
void pop() {
    log("pop");
    if (s > 0) {
        s--;
    }
}
```

Errors:

- Array out of bounds.
- top returns an invalid value!

## Example – stack (Cont'd)

```
#define TLEN 100
int main () {
    int v, action;
    for (int i = 0; i < TLEN; i++) {
        action = nondet_int();
        assume ((action >= 0) && (action <= 2));
        switch (action) {
            case 0:
                v = top();
                break;
            case 1:
                v = nondet_int();
                push(v);
                break;
            case 2:
                pop();
                break;
        }
    }
}
```

# Algorithm For Swarm Bounded Model Checking

**Input:** program  $p$ , set  $F$  of features

- 1: **while** budget allows
- 2:     **for** all processors available
- 3:         Pick a random set  $F_i \subseteq F$
- 4:         Build  $sp$  by replacing `log` statements in  $p$  for  $F_i$  with  
           `assume(false)` and Remove other `log` statements from
- 5:         Propagate assumptions and slice  $sp$
- 6:         Perform BMC on  $sp$

*Intuition:* By omitting features in a program's execution, we can produce smaller and more easily checked SAT instances.

## Example – Omitting the top feature in the stack

```
int top() {
    __CPROVER_assume(false);
    return stack[s];
}
void push(int i) {
    stack[s++] = i;
}
void pop() {
    if (s > 0) {
        s--;
    }
}
```

# Experimental results

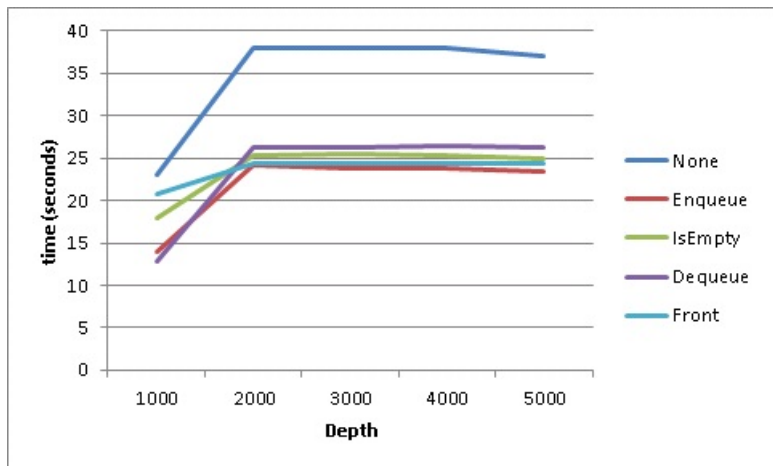
We used CBMC version 4.0 for bounded model checking on a four-processor Intel 2.8GHz system with 8 GB RAM on some data structures.

Early experiment results suggest that the swarm bounded model checking are faster in detecting counterexamples.

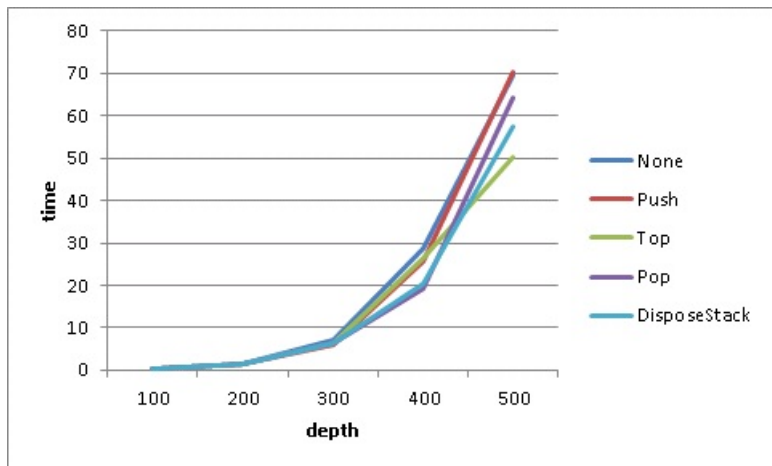
# Result of swarm bounded model checking on stack example

Omitted Feature	Time without Slicing (Seconds)	Time with Slicing (Seconds)	Verification Status
–	325	49	Counterexample
push	40	4	Verified
pop	101	14	Counterexample
top	291	48	Counterexample

# Experimental Results – Array Queue



# Experimental Results – Stack List





## Related work

- Swarm Verification:
  - Dwyer et al. *Parallel randomized state-space search*. ICSE 2007.
  - Holzmann et al. *Swarm verification techniques*. TSE 2010.
- Swarm Testing: Groce et al. *Swarm Testing*.
- Conditional Verification: Beyer et al. *Conditional Model Checking*. University of Passau Tech report, (2011).

# Conclusion

- By omitting features in a program's execution, we can produce smaller and more easily checked SAT instances, while often preserving at least one counterexample trace.
- Early results suggest that bounded model checking with feature omission outperforms the traditional BMC in returning a counterexample. However, we need to apply swarm BMC to larger, more realistic examples that challenge the abilities of current BMC tools.