Ant Colony Optimization for Testing Concurrent Systems: Analysis of Scalability

Francisco Chicano and Enrique Alba
Concurrent software is difficult to test ...
... and it is in the heart of a lot of critical systems

Techniques for proving the correctness of concurrent software are required
Model checking → fully automatic
Traditional techniques for this purpose have problems with large models
We analyze here the scalability of a new proposal: ACOhg-mc
Objective: Prove that model $M$ satisfies the property $f$: $M \models f$

HSF-SPIN: the property $f$ is an LTL formula

Model $M$

LTL formula $\neg f$
(never claim)

Intersection Büchi automaton
• **Objective:** Prove that model $M$ satisfies the property $f$: $M \models f$

• **HSF-SPIN:** the property $f$ is an LTL formula
Explicit State Model Checking

- **Objective:** Prove that model $M$ satisfies the property $f$: $M \models f$
- **HSF-SPIN:** the property $f$ is an LTL formula

![Diagram of model $M$, LTL formula $\neg f$ (never claim), and Intersection Büchi automaton]

Using Nested-DFS
State Explosion Problem

- Number of states very large even for small models

- Example: Dining philosophers with \( n \) philosophers \( \rightarrow 3^n \) states
  20 philosophers \( \rightarrow 1039 \) GB for storing the states

- Solutions: collapse compression, minimized automaton representation, bitstate hashing, partial order reduction, symmetry reduction

- Large models cannot be verified but errors can be found
The search for errors can be directed by using heuristic information.

Different kinds of heuristic functions have been proposed in the past:

- Formula-based heuristics
- Structural heuristics
- Deadlock-detection heuristics
- State-dependent heuristics
Safety and Liveness Properties

Safety property

\[ \forall \sigma \in S^\omega : \sigma \not\models \mathcal{P} \Rightarrow (\exists i \geq 0 : \forall \beta \in S^\omega : \sigma_i \beta \not\models \mathcal{P}) \]

- Counterexample \( \equiv \) path to accepting state
- Graph exploration algorithms can be used: DFS and BFS

Liveness property

\[ \forall \alpha \in S^* : \exists \beta \in S^\omega , \alpha \beta \models \mathcal{P} \]

- Counterexample \( \equiv \) path to accepting cycle
- It is not possible to apply DFS or BFS
Optimization/Search Techniques

- Exact
- Approximated

Exact
- Based on Calculus
  - Newton
  - Gradient
- Enumeratives
  - Depth First Search
  - Branch and Bound

Approximated
- Ad Hoc Heuristics
- Metaheuristics
  - Trayectory-based
    - SA
    - VNS
    - TS
  - Population-based
    - EA
    - ACO
    - PSO
Ant Colony Optimization (ACO) metaheuristic is inspired by the foraging behaviour of real ants.

ACO Pseudo-code:

```plaintext
procedure ACOMetaheuristic
    ScheduleActivities
        ConstructAntsSolutions
        UpdatePheromones
    DaemonActions // optional
end ScheduleActivities
end procedure
```
ACO: Construction Phase

- The ant selects its next node **stochastically**

- The probability of selecting one node depends on the **pheromone trail** and the **heuristic value** (optional) of the edge/node

- The ant stops when a complete solution is built
ACO: Pheromone Update

- Pheromone update
  - During the construction phase
    \[ \tau_{ij} \leftarrow (1 - \xi)\tau_{ij} \quad \text{with} \quad 0 \leq \xi \leq 1 \]
  - After the construction phase
    \[ \tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \Delta\tau_{ij}^{bs} \quad \text{with} \quad 0 \leq \rho \leq 1 \]

- Trail limits (particular of MMAS)
  - Pheromones are kept in the interval \([\tau_{min}, \tau_{max}]\)
    \[ \tau_{max} = \frac{Q}{\rho} \]
    \[ \tau_{min} = \frac{\tau_{max}}{a} \]
ACOhg: Huge Graphs Exploration

The length of the ant paths is limited by $\lambda_{\text{ant}}$

What if…?

Starting nodes for path construction change

- Initial node
- Objective node
- After $\sigma_s$ steps
- Second stage
- Third stage
• The search is an alternation of two phases

- **First phase:** search for accepting states
- **Second phase:** search for cycles from the accepting states

**ACOhg-mc Pseudocode**

```python
repeat
  accept = acohg1.findAcceptingStates(); {First phase}
  for node in accept do
    acohg2.findCycle(node); {Second phase}
    if acohg2.cycleFound() then
      return acohg2.acceptingPath();
    end if
  end for
  acohg1.insertTabu(accept);
until empty(accept)
return null;
```
The search is an alternation of two phases

- First phase: search for accepting states
- Second phase: search for cycles from the accepting states

**ACOhg-mc Pseudocode**

```plaintext
1: repeat
2:    acpt = acohg1.findAcceptingStates(); {First phase}
3:    for node in acpt do
4:       acohg2.findCycle(node); {Second phase}
5:    if acohg2.cycleFound() then
6:       return acohg2.acceptingPath();
7:    end if
8: end for
9: acohg1.insertTabu(acpt);
10: until empty(acpt)
11: return null;
```
ACOhg-mc

- The search is an alternation of two phases

  - **First phase**: search for accepting states
  - **Second phase**: search for cycles from the accepting states

### ACOhg-mc Pseudocode

```
1: repeat
2:   acpt = acohg1.findAcceptingStates(); {First phase}
3:   for node in acpt do
4:     acohg2.findCycle(node); {Second phase}
5:     if acohg2.cycleFound() then
6:       return acohg2.acceptingPath();
7:     end if
8:   end for
9:   acohg1.insertTabu(acpt);
10: until empty(acpt)
11: return null;
```
We used 4 scalable Promela models for the experiments

<table>
<thead>
<tr>
<th>Model</th>
<th>LoC</th>
<th>Processes</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>marriersj</td>
<td>64</td>
<td>j+1</td>
<td>deadlock</td>
</tr>
<tr>
<td>elevj</td>
<td>191</td>
<td>j+3</td>
<td>$\square(p \rightarrow \Diamond q)$</td>
</tr>
<tr>
<td>giopj</td>
<td>740</td>
<td>j+6</td>
<td>deadlock and $\square(p \rightarrow \Diamond q)$</td>
</tr>
<tr>
<td>phi$\overline{e}$j</td>
<td>57</td>
<td>j+1</td>
<td>deadlock and $\square(p \rightarrow \Diamond q)$</td>
</tr>
</tbody>
</table>

Parameters for ACOhg-mc

<table>
<thead>
<tr>
<th>Parameter</th>
<th>msteps</th>
<th>colszie</th>
<th>$\lambda_{ant}$</th>
<th>$\sigma_s$</th>
<th>$\xi$</th>
<th>$a$</th>
<th>$\rho$</th>
<th>$\alpha$</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st phase</td>
<td>100</td>
<td>10</td>
<td>40</td>
<td>4</td>
<td>0.7</td>
<td>5</td>
<td>0.2</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>2nd phase</td>
<td></td>
<td>20</td>
<td>4</td>
<td>4</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Formula-based and finite state machine heuristics

ACOhg-mc implemented in HSF-SPIN

100 independent executions
**Promela Models**

- We used 4 scalable Promela models for the experiments

<table>
<thead>
<tr>
<th>Model</th>
<th>LoC</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>marriersj</td>
<td>64</td>
<td>j+1 deadlock</td>
</tr>
<tr>
<td>elevj</td>
<td>191</td>
<td>j+3 □(p → ◊q)</td>
</tr>
<tr>
<td>giopj</td>
<td>740</td>
<td>j+6 deadlock and □(p → ◊q)</td>
</tr>
<tr>
<td>phi j</td>
<td>57</td>
<td>j+1 deadlock and □(p → ◊q)</td>
</tr>
</tbody>
</table>

- Parameters for ACOhg-mc

<table>
<thead>
<tr>
<th>Parameter</th>
<th>msteps</th>
<th>colsize</th>
<th>λant</th>
<th>σs</th>
<th>ξ</th>
<th>a</th>
<th>ρ</th>
<th>α</th>
<th>β</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st phase</td>
<td>100</td>
<td>10</td>
<td>40</td>
<td>4</td>
<td>0.7</td>
<td>5</td>
<td>0.2</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>2nd phase</td>
<td>20</td>
<td>4</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Formula-based and finite state machine heuristics
- ACOhg-mc implemented in HSF-SPIN
- 100 independent executions
Efficacy

Models & parameters

Results

Efficacy

Models

ACOhg-mc NDFS

ACOhg-mc
NDFS

phi-r
giop-r
elev-r
phi-d
marriers-d

0 4 8 12 16 20 24 28
n
Analysis of Scalability: Memory

<table>
<thead>
<tr>
<th>Models &amp; parameters</th>
<th>Results</th>
</tr>
</thead>
</table>

ACO

- marriers-d
- giop-d
- phi-d
- elev-r
- giop-r
- phi-r

Graph showing memory usage for different models and parameters.
Analysis of Scalability: Memory

Introduction

Results

Models & parameters

Algorithmic Proposal

Experiments

Conclusions & Future Work

NDFS

Memory (KB)

n

marriers-d  giop-d  phi-d  elev-r  giop-r  phi-r

META’08, Hammamet, Tunisia, October 29-31, 2008
Analysis of Scalability: Length

ACOhg-me

Graph showing the relationship between Length and n with different markers and lines.
Analysis of Scalability: Length

NDFS

models & parameters

results

introduction

background

algorithmic proposal

experiments

conclusions & future work
Analysis of Scalability: CPU time

Models & parameters  Results

ACOhg-mc

Time (ms)

marriers-d  giop-d  phi-d  elev-r  giop-r  phi-r

n

2  6  10  14  18  22  26  30

0  5000  10000  15000  20000  25000  30000  35000
Analysis of Scalability: CPU time

![Graph showing CPU time vs. n for various models and parameters]

Models & parameters

Results

NDFS
Conclusions

• ACOhg-mc is able to find errors in large models for which NDFS fails

• The memory required by ACOhg-mc for the search is small and grows very slowly

• The length of the error trails increases linearly in most of the cases

• Although ACOhg-mc is not always the fastest algorithm, the time required is small

Future Work

• Analysis of parameterization for reducing the parameters

• Include ACOhg-mc into Java PathFinder for finding errors in Java programs

• Combine ACOhg-mc with techniques for reducing the memory required for the search such as partial order reduction
Ant Colony Optimization for Testing concurrent Systems: Analysis of Scalability

Thanks for your attention !!!