Finding Safety Errors with ACO

Enrique Alba and Francisco Chicano
Motivation

• Nowadays software is very complex

• An error in a software system can imply the loss of lot of money …

… and even human lifes

• Techniques for proving the correctness of the software are required

• Model checking → fully automatic
Objective: Prove that model $M$ satisfies the property $\phi$: $M \models \phi$

SPIN: the property $\phi$ is an LTL formula

Model $M$  

LTL formula $\neg \phi$

Intersection Büchi automaton

Explicit State Model Checking
• **Objective:** Prove that model $M$ satisfies the property $f$: $M \models f$

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

**SPIN:** the property $f$ is an LTL formula

Model $M$  

LTL formula $\neg f$

Intersection Büchi automaton

Using Nested-DFS
Safety Properties

- Safety properties are those expressed by an LTL formula of the form:

\[ f = \Box p \]

where \( p \) is a past formula

- Finding one counterexample \( \equiv \) finding one accepting state

Classical algorithms for graph exploration can be used: DFS and BFS
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 heuristics using algorithms like $A^*$, $IDA^*$, $WA^*$ and Best-First.

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

- **Formula-based heuristics**
- **Structural heuristics**
- **Deadlock-detection heuristics**
- **State-dependent heuristics**
Metaheuristic Algorithms

- Designed to solve optimization problems
  - Maximize or minimize a given function: the fitness function
- They can find “good” solutions with a “reasonable” amount of resources
Metaheuristics Classification

**Single solution**
- Greedy Randomized Adaptive Search Procedure
- Iterated Local Search
- Variable Neighborhood Search
- Tabu Search
- Simulated Annealing
- Iterative Improvement
- Guided Local Search

**Population**
- Estimation of Distribution Algorithms
- Evolutionary Computation
- Scatter Search
- Ant Colony Optimization
- Particle Swarm Optimization
Metaheuristics Classification

Single solution
- Greedy Randomized Adaptive Search Procedure
- Tabu Search
- Simulated Annealing
- Iterated Local Search
- Iterative Improvement
- Variable Neighborhood Search
- Guided Local Search

Population
- Estimation of Distribution Algorithms
- Evolutionary Computation
- Scatter Search
- Ant Colony Optimization
- Particle Swarm Optimization

Genetic Algorithms
- Alba & Troya, 1996
- Godefroid & Khurshid, 2002, 2004
Ant Colony Optimization (ACO) metaheuristic is inspired by the foraging behaviour of real ants.

ACO Pseudo-code:

```
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

- 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
    \[ \tau_{ij} \leftarrow \rho\tau_{ij} + \Delta\tau_{ij}^{bs} \quad \text{with} \quad 0 \leq \rho \leq 1 \]

• Trail limits (particular of MMAS)
  ➢ Pheromones are kept in the interval \([\tau_{\text{min}}, \tau_{\text{max}}]\)
    \[ \tau_{\text{max}} = \frac{Q}{1 - \rho} \quad \tau_{\text{min}} = \frac{\tau_{\text{max}}}{a} \]
ACOhg: Motivation

• Existing ACO models cannot be applied to the search for errors in concurrent programs
  ➢ The graph is very large, the construction of a complete solution could require too much time and memory
  ➢ In some models the number of nodes of the graph is used for computing the initial pheromone values

• We need a new model for tackling these problems: ACOhg (ACO for Huge Graphs)
  ➢ Constructs the ant paths and updates the pheromone values in the same way as the traditional models
  ➢ Allows the construction of partial solutions
  ➢ Allows the exploration of the graph using a bounded amount of memory
  ➢ The pheromone matrix is never completely stored
ACOhg: Huge Graphs Exploration

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

- Initial node
- Objective node
- What if...?

$\lambda_{\text{ant}}$
ACOhg: Huge Graphs Exploration

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

Expansion Technique: $\lambda_{ant}$ changes

After $\sigma_1$ steps

$\lambda_{ant} = \lambda_{ant} + \delta_1$
ACOhg: Huge Graphs Exploration

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

Missionary Technique: starting nodes for path construction change

What if…?

Objective node

Two alternatives

Initial node

After $\sigma_s$ steps

Second stage

Third stage

London, United Kingdom, July 7-11, 2007
• The number of pheromone trails increases during the search
• This leads to memory problems
• We must remove some pheromone trails from memory

Remove pheromone trails $\tau_{ij}$ below a given threshold $\tau_\theta$

In the missionary technique, remove all pheromone trails after one stage

Pheromones

Steps

Pheromones

Stage

Pheromones

Steps
ACOhg: Fitness Function

- The fitness function must be able to evaluate partial solutions
- Penalties are added for partial solutions and solutions with cycles

**Complete solution**

\[ p = 0 \]

**Partial solution without cycle**

\[ p = p_p \]

\[ p = p_p + \frac{\lambda_{\text{ant}}}{\lambda_{\text{ant}}} l \]

**Partial solution with cycle**

Total penalty

Penalty constant for partial solutions

Penalty constant for solutions with cycles

Path length
Promela Models

- We selected 5 Promela models for the experiments

<table>
<thead>
<tr>
<th>Model</th>
<th>LoC</th>
<th>States</th>
<th>Processes</th>
<th>Safety Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>giop22</td>
<td>717</td>
<td>unknown</td>
<td>11</td>
<td>Deadlock</td>
</tr>
<tr>
<td>marriers4</td>
<td>142</td>
<td>unknown</td>
<td>5</td>
<td>Deadlock</td>
</tr>
<tr>
<td>needham</td>
<td>260</td>
<td>18242</td>
<td>4</td>
<td>LTL formula</td>
</tr>
<tr>
<td>phi16</td>
<td>34</td>
<td>43046721*</td>
<td>17</td>
<td>Deadlock</td>
</tr>
<tr>
<td>pots</td>
<td>453</td>
<td>unknown</td>
<td>8</td>
<td>Deadlock</td>
</tr>
</tbody>
</table>

* Theoretical result

- For all except needham, the states do not fit into the main memory of the computer
• The ACOhg model was implemented inside the MALLBA library and then included into the HSF-SPIN model checker

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steps</td>
<td>100</td>
<td>ξ</td>
<td>0.5</td>
</tr>
<tr>
<td>Colony size</td>
<td>10</td>
<td>α</td>
<td>5</td>
</tr>
<tr>
<td>λ&lt;sub&gt;ant&lt;/sub&gt;</td>
<td>10</td>
<td>ρ</td>
<td>0.8</td>
</tr>
<tr>
<td>σ&lt;sub&gt;s&lt;/sub&gt;</td>
<td>2</td>
<td>α</td>
<td>1.0</td>
</tr>
<tr>
<td>s</td>
<td>10</td>
<td>β</td>
<td>2.0</td>
</tr>
</tbody>
</table>

• Fitness function: length of the path + heuristic + penalty for partial solutions

• Two variants: using no heuristic (ACOhg-b) and using it (ACOhg-h)

• Machine: Pentium 4 at 2.8 GHz with 512 MB
Results I: Efficacy

- We compare the results of ACOhg algorithms against state-of-the-art model checker algorithms: DFS, BFS, A*, and BF

Which algorithm finds errors?

<table>
<thead>
<tr>
<th>Models</th>
<th>BFS</th>
<th>DFS</th>
<th>A*</th>
<th>BF</th>
<th>ACOhg</th>
</tr>
</thead>
<tbody>
<tr>
<td>giop22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><img src="ant" alt="Ant" /></td>
</tr>
<tr>
<td>needham</td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
</tr>
<tr>
<td>phil6</td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
</tr>
<tr>
<td>pots</td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
</tr>
<tr>
<td>marriers4</td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
</tr>
<tr>
<td>marriers20</td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
<td><img src="ant" alt="Ant" /></td>
</tr>
</tbody>
</table>

- ACOhg algorithms are the only ones that are able to find errors in very large models (marriers20).
## Results II: Details

<table>
<thead>
<tr>
<th>Models</th>
<th>Measurements</th>
<th>BFS</th>
<th>DFS</th>
<th>ACOhg-b</th>
<th>$A^*$</th>
<th>BF</th>
<th>ACOhg-h</th>
</tr>
</thead>
<tbody>
<tr>
<td>giop22</td>
<td>hit rate</td>
<td>0/1</td>
<td>1/1</td>
<td>100/100</td>
<td>1/1</td>
<td>1/1</td>
<td>100/100</td>
</tr>
<tr>
<td></td>
<td>len (states)</td>
<td>-</td>
<td>112.00</td>
<td>45.80</td>
<td>44.00</td>
<td>44.00</td>
<td>44.20</td>
</tr>
<tr>
<td></td>
<td>mem (KB)</td>
<td>-</td>
<td>3945.00</td>
<td>4814.12</td>
<td>417792.00</td>
<td>2873.00</td>
<td>4482.12</td>
</tr>
<tr>
<td></td>
<td>exp (states)</td>
<td>-</td>
<td>220.00</td>
<td>1048.52</td>
<td>83758.00</td>
<td>168.00</td>
<td>1001.78</td>
</tr>
<tr>
<td></td>
<td>cpu (ms)</td>
<td>-</td>
<td>30.00</td>
<td>113.60</td>
<td>16440.00</td>
<td>10.00</td>
<td>112.10</td>
</tr>
<tr>
<td>marriers4</td>
<td>hit rate</td>
<td>0/1</td>
<td>0/1</td>
<td>57/100</td>
<td>0/1</td>
<td>1/1</td>
<td>84/100</td>
</tr>
<tr>
<td></td>
<td>len (states)</td>
<td>-</td>
<td>-</td>
<td>92.18</td>
<td>-</td>
<td>108.00</td>
<td>86.65</td>
</tr>
<tr>
<td></td>
<td>mem (KB)</td>
<td>-</td>
<td>-</td>
<td>5917.91</td>
<td>-</td>
<td>41980.00</td>
<td>5811.43</td>
</tr>
<tr>
<td></td>
<td>exp (states)</td>
<td>-</td>
<td>-</td>
<td>2045.81</td>
<td>-</td>
<td>9193.00</td>
<td>1915.30</td>
</tr>
<tr>
<td></td>
<td>cpu (ms)</td>
<td>-</td>
<td>-</td>
<td>257.19</td>
<td>-</td>
<td>190.00</td>
<td>233.33</td>
</tr>
<tr>
<td>neddham</td>
<td>hit rate</td>
<td>1/1</td>
<td>1/1</td>
<td>100/100</td>
<td>1/1</td>
<td>1/1</td>
<td>100/100</td>
</tr>
<tr>
<td></td>
<td>len (states)</td>
<td>5.00</td>
<td>11.00</td>
<td>6.39</td>
<td>5.00</td>
<td>10.00</td>
<td>6.12</td>
</tr>
<tr>
<td></td>
<td>mem (KB)</td>
<td>23552.00</td>
<td>62464.00</td>
<td>5026.36</td>
<td>19456.00</td>
<td>4149.00</td>
<td>1865.10</td>
</tr>
<tr>
<td></td>
<td>exp (states)</td>
<td>1141.00</td>
<td>11203.00</td>
<td>100.21</td>
<td>811.00</td>
<td>12.00</td>
<td>87.17</td>
</tr>
<tr>
<td></td>
<td>cpu (ms)</td>
<td>1110.00</td>
<td>18880.00</td>
<td>262.00</td>
<td>810.00</td>
<td>20.00</td>
<td>229.50</td>
</tr>
<tr>
<td>phi16</td>
<td>hit rate</td>
<td>0/1</td>
<td>0/1</td>
<td>100/100</td>
<td>1/1</td>
<td>1/1</td>
<td>100/100</td>
</tr>
<tr>
<td></td>
<td>len (states)</td>
<td>-</td>
<td>-</td>
<td>31.11</td>
<td>17.00</td>
<td>81.00</td>
<td>23.08</td>
</tr>
<tr>
<td></td>
<td>mem (KB)</td>
<td>-</td>
<td>-</td>
<td>10905.60</td>
<td>288.10</td>
<td>10240.00</td>
<td>10680.32</td>
</tr>
<tr>
<td></td>
<td>exp (states)</td>
<td>-</td>
<td>-</td>
<td>882.08</td>
<td>33.00</td>
<td>893.00</td>
<td>587.53</td>
</tr>
<tr>
<td></td>
<td>cpu (ms)</td>
<td>-</td>
<td>-</td>
<td>289.40</td>
<td>10.00</td>
<td>40.00</td>
<td>243.80</td>
</tr>
<tr>
<td>pots</td>
<td>hit rate</td>
<td>1/1</td>
<td>1/1</td>
<td>49/100</td>
<td>1/1</td>
<td>1/1</td>
<td>90/100</td>
</tr>
<tr>
<td></td>
<td>len (states)</td>
<td>5.00</td>
<td>14.00</td>
<td>5.73</td>
<td>5.00</td>
<td>7.00</td>
<td>5.14</td>
</tr>
<tr>
<td></td>
<td>mem (KB)</td>
<td>57344.00</td>
<td>12288.00</td>
<td>9304.67</td>
<td>57344.00</td>
<td>6389.00</td>
<td>6974.56</td>
</tr>
<tr>
<td></td>
<td>exp (states)</td>
<td>2037.00</td>
<td>1966.00</td>
<td>176.47</td>
<td>1257.00</td>
<td>695.00</td>
<td>110.18</td>
</tr>
<tr>
<td></td>
<td>cpu (ms)</td>
<td>4190.00</td>
<td>140.00</td>
<td>441.63</td>
<td>6640.00</td>
<td>50.00</td>
<td>319.49</td>
</tr>
</tbody>
</table>
Results III: Graphical Comparison

- Error trail length vs. memory graph

ACOhg algorithms require less memory than BFS

They also get shorter (better) error trails than DFS

- In general, unlike exhaustive algorithms, ACOhg algorithms keep all the results in a good performance region (high accuracy and efficiency)
Previous Results with Metaheuristics

• **GA** is the previous metaheuristic algorithm applied to this problem

• Godefroid & Khurshid (2002), found errors in phi17 and needham models with GA

• To the best of our knowledge, this is the most recent result for this problem using metaheuristics

<table>
<thead>
<tr>
<th>Model</th>
<th>Algorithm</th>
<th>Hit (%)</th>
<th>Time (s)</th>
<th>Mem. (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>phi17</td>
<td>GA</td>
<td>52</td>
<td>197.00</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ACOHg-h</td>
<td>100</td>
<td>0.28</td>
<td>11274</td>
</tr>
<tr>
<td>needham</td>
<td>GA</td>
<td>3</td>
<td>3068.00</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ACOHg-h</td>
<td>100</td>
<td>0.23</td>
<td>4865</td>
</tr>
</tbody>
</table>

• The results state that ACOHg has **higher efficacy and efficiency** than GA (even taking into account the differences in the machines)

• But we cannot do a fair comparison because the models and the model checkers are different (Verisoft against HSF-SPIN)
Conclusions

• ACOhg is able to **outperform state-of-the-art algorithms used nowadays in current model checkers for finding safety errors**

• ACOhg is able to explore **really large concurrent models for which traditional model checking techniques fail**

• This represents a **promising starting point for the use of metaheuristic algorithms in model checking and an interesting subject in SBSE**

Future Work

• Combine ACOhg algorithms with other techniques for reducing the amount of memory: **Partial Order Reduction and Symmetry Reduction** (in progress)

• Include ACOhg into **JavaPathFinder** for finding errors in Java programs (in progress)

• Parallel implementation of ACOhg for this problem (parallel model checkers)
Finding Safety Errors with ACO

Thanks for your attention !!!!