An Alternative Structure for SPIN?

PJA de Villiers
Department of Computer Science
University of Stellenbosch
Stellenbosch
South Africa
Introduction

- Project goals:

  1. Validation of a microkernel

  2. Learn about validators (implement one specifically to validate the microkernel)

- **Focus of this talk:** experience that may have some value for SPIN
The validation process

1. Generate a state

2. Determine whether it is a new state

3. If not a new state, then
   (a) a loop
   (b) a revisited state

4. Analyse the state
What is different?

• State generation
  – Code generated for abstract machine

• State storage
  – State cache with replacement + compaction

• State analysis
  – CTL for correctness claims
  – Nested subformulae handled as “subproblems”

• Highly modular design
State generation

Measurements (interpolated state generator):

<table>
<thead>
<tr>
<th></th>
<th>Philos</th>
<th>Elevator</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans</td>
<td>2338593</td>
<td>9086599</td>
<td>9908147</td>
</tr>
<tr>
<td>Time (secs)</td>
<td>123.52</td>
<td>439.81</td>
<td>450.07</td>
</tr>
<tr>
<td>Trans/sec</td>
<td>18933</td>
<td>20660</td>
<td>22015</td>
</tr>
</tbody>
</table>

Measurements (SPIN):

<table>
<thead>
<tr>
<th></th>
<th>pftp1</th>
<th>leader</th>
<th>pftp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans</td>
<td>1301620</td>
<td>185032</td>
<td>1223060</td>
</tr>
<tr>
<td>Time (secs)</td>
<td>1:08.38</td>
<td>9.29</td>
<td>47.98</td>
</tr>
<tr>
<td>Trans/sec</td>
<td>19035</td>
<td>19917</td>
<td>25491</td>
</tr>
</tbody>
</table>

- Abstract machine: validator more understandable, fast enough

- Difficult to isolate the effect of interpretation
State storage

- How costly is state compaction?

<table>
<thead>
<tr>
<th>Module</th>
<th>% of time</th>
</tr>
</thead>
<tbody>
<tr>
<td>State generator</td>
<td>50.43</td>
</tr>
<tr>
<td>Scheduler</td>
<td>10.90</td>
</tr>
<tr>
<td>Storage</td>
<td>21.63</td>
</tr>
<tr>
<td>Compaction</td>
<td>15.17</td>
</tr>
<tr>
<td>Other</td>
<td>1.87</td>
</tr>
</tbody>
</table>

- Substantial reduction in state size. For example, states of a rather complex model (the scheduler of the microkernel) fit in 16 bytes)
Benefits of state compaction...

- cache performance is improved
  - fewer state replacements (more states fit into memory)
  - hash function less costly to compute (state is smaller)

- Exhaustive state exploration possible for larger models
Computing compact states...

VAR c: (red, green, blue, white, black);
   b: BOOLEAN;
   p: RECORD x, y: 0..149 END;

- Just enough space allocated per variable

- Compacted state $S$ is computed as
  \[ c + 5 \cdot (b + 2 \cdot (p.x + 150 \cdot p.y)) \]
  \[ = c + 5 \cdot b + 5 \cdot 2 \cdot p.x + 5 \cdot 2 \cdot 150 \cdot p.y \]

- Each variable $z$ has two associated “shield” factors $z_{lo}$ and $z_{hi}$ ($b_{lo} = 5$ and $b_{hi} = 5 \cdot 2$)
Extracting values from compacted states...

- The value of variable \( z = (S \mod z_{hi}) \div z_{lo} \)

- When the value of variable \( z \) changes to \( z' \), the updated state is \( S' = S + z_{lo} \cdot (z' - z) \)

- The factors \( z_{lo} \) and \( z_{hi} \) are computed at compile time

- The cost of variable lookup is 2 operations (mod, divide)

- The cost of modifying a variable is 3 operations (add, subtract, multiply)
Handling nested temporal subformulae... 

- Example: $AG(\alpha \Rightarrow AF\beta)$

- Key idea: assume $AF\beta$ is true and postpone evaluation until it is more convenient

- Subset of states computed where further evaluation is needed (states where $\alpha$ is true)

- No storage needed for values of nested subformulæ
Avoiding unnecessary work...

\[ EF\alpha \equiv \alpha \lor EXEF\alpha \]
\[ AF\alpha \equiv \alpha \lor AXAF\alpha \]
\[ EG\alpha \equiv \alpha \lor EXEG\alpha \]
\[ AG\alpha \equiv \alpha \lor AXAG\alpha \]
\[ E(\alpha \cup \beta) \equiv \beta \lor (\alpha \land EXE(\alpha \cup \beta)) \]
\[ A(\alpha \cup \beta) \equiv \beta \lor (\alpha \land AXA(\alpha \cup \beta)) \]

- use the cache to avoid unnecessary work

- cache invariant: the current subformula holds for all states in the cache

- requirement: no state is entered in the cache before it is known that the current subformula holds for it
Structure of validator

- state analyser
- state generator
  - current state
- state storage
  - hash table (cache)
<table>
<thead>
<tr>
<th>Module</th>
<th>Lines</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>State generator</td>
<td>803</td>
<td>17132</td>
</tr>
<tr>
<td>Scheduler</td>
<td>221</td>
<td>3040</td>
</tr>
<tr>
<td>Storage</td>
<td>142</td>
<td>1568</td>
</tr>
<tr>
<td>Compaction</td>
<td>161</td>
<td>2196</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1327</strong></td>
<td><strong>23936</strong></td>
</tr>
</tbody>
</table>
Conclusions

- Interpretation simplifies the validator and is probably efficient enough

- Compaction does not cost much and makes a state cache very effective

- The main functions of a validator can be implemented in three relatively independent modules without making it grossly inefficient. These modules are:
  - state generation
  - state storage and compaction
  - state analysis