MAGE
the Lazy Symbolic Game-Based Safety Checker

Adam Bakewell
University of Birmingham, U.K.
a.bakewell@cs.bham.ac.uk

MAGE\textsuperscript{1} is a new experimental software model checking tool based on game semantics. The game-based approach has several inherent advantages:

**Full abstraction** Every program behaviour is represented in the model and every represented behaviour is exhibited by the program. Thus all errors can be detected and error detection implies a fault.

**Compositionality** The model of a program can be built from the models of sub-programs. Open sub-programs can be modelled and verified independently, implying the potential for greater scalability and lighter re-analysis.

**Black-box modelling** The model only records observable actions. Actions on the local state are hidden. The reduction in transitions in the model reduces memory requirements and search space size.

Existing work shows that game models can be given effective algorithmic representations and used as a basis for model checking [1]. The first game-based model checker could verify some challenging programs such as sorting and certain kinds of abstract data types [2]. More recently a counterexample-guided refinement technique was adapted to game models [3], and a prototype tool called GAMECHECKER was developed [4].

However, these efforts do a whole model extraction followed by a checking phase. When checking unsafe programs this involves a lot of wasted work as only a fraction of the model is searched. Moreover, naively composing the models of program components involves discarding vast portions that encode the unused behaviours. MAGE tackles these problems and more by introducing (or adapting) the following techniques to game models.

**Symbolic models** Representing the model by equations rather than explicit states and transitions allows a symbolic model to be built implicitly in linear time. Parts of the model are made explicit by tracing the next-state function provided by a symbol. Symbol composition thereby avoids generating unused traces in sub-models.

**Lazy model checking** The checker only generates parts of the model immediately before it tests them for safety and only stores them as long as they are part of some trace in the search queue. Thus the explicit model is only partly generated when the program is unsafe and in all cases only some fraction of it is stored at any moment.

\textsuperscript{1} Download it from \url{http://www.cs.bham.ac.uk/axb/games/mage/}.
Approximation The explicit model size, and hence search time, is greatly reduced by using safe over-approximations. Concretely, Mage replaces individual numbers by ranges. This may lead to the discovery of “false positives” by sacrificing one direction of the full abstraction property.

Iterated refinement Potential errors that cannot be validated as “true positives” are used to create more precise approximations and the model is re-searched. This technique is enabled by introducing “grey-box” game models which release information about how hidden variables are used.

Approximated counterexample validation Genuine unsafety can often be recognised long before the refinement loop reaches precise (concrete) domains. This is also enabled by the grey-box information, in this case revealing whether the approximation has produced an error trace that combines potentially incompatible sub-traces.

Mage implements these techniques for programs written in the call-by-name Idealized Algol language, allowing direct comparison with the earlier game-based model checkers. For example, Table 1 shows the time taken to detect situations leading to overflows in integer stacks of different sizes: the Mage time is essentially linear but GameChecker is exponential because it is dominated by model building. Both benefit from data abstractions recognizing that the stacked integers are irrelevant to the overflow property. The third column gives the time taken by the Blast model checker (for C programs) on the same problem. Blast [5] also uses abstraction, refinement and laziness but it is not a compositional game-based tool; rather, it represents the state of the art in model checking based on predicate abstraction.

Table 1. Stack overflow detection tests.

<table>
<thead>
<tr>
<th>Stack Size</th>
<th>Mage (sec)</th>
<th>GameChecker (sec)</th>
<th>Blast 2.0 (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.1</td>
<td>10.1</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>27.5</td>
<td>3.3</td>
</tr>
<tr>
<td>8</td>
<td>0.2</td>
<td>112.6</td>
<td>4.6</td>
</tr>
<tr>
<td>16</td>
<td>0.4</td>
<td>780.7</td>
<td>7.8</td>
</tr>
<tr>
<td>32</td>
<td>1.2</td>
<td>12.268.1</td>
<td>17.3</td>
</tr>
<tr>
<td>64</td>
<td>3.9</td>
<td>over 7 hours</td>
<td>43.7</td>
</tr>
<tr>
<td>128</td>
<td>13.9</td>
<td>-</td>
<td>145.3</td>
</tr>
<tr>
<td>256</td>
<td>54.8</td>
<td>-</td>
<td>space exhausted</td>
</tr>
</tbody>
</table>

References