Learning Finite-State Machines: Conserving Fitness Evaluations by Marking Used Transitions

Daniil Chivilikhin and Vladimir Ulyantsev
National Research University of IT, Mechanics and Optics
St. Petersburg, Russia

Machine Learning Applications in Software Engineering @ ICMLA’13
December 6, 2013
Outline

• Motivation and scope
• Proposed idea
• Experiments
  – Artificial Ant problem
  – Test-based EFSEM induction
• Conclusion
Motivation and scope
Motivation: Reliable software

• Systems with high cost of failure
  – Energy industry
  – Aircraft industry
  – Space industry
  – ...

• We want to have **reliable software**
  – Testing is not enough
  – **Verification** is needed
Verification

• Checking temporal rules (e.g. $LTL$)
• Software verification can be harder than software development
• Need to make software that satisfies $LTL$-specification by design
• How?
Model-driven development

- Automated software engineering
- Model-driven development
Model-driven development

Finite-state machine

- Software specification
- Model
- Code
Finite-State Machine

- $\Sigma$ – set of input events
- $\Delta$ – set of output actions
- $\delta: S \times \Sigma \rightarrow S$ – transition function
- $\lambda: S \times \Sigma \rightarrow \Delta$ – actions function
Automata-based programming

Design programs with complex behavior as automated-controlled objects

Automated-controlled object

Finite-state machine

Events

Output actions

Controlled object

Actions

Learning FSMs: Conserving Fitness Evaluations
Automata-based programming: advantages

- Model before programming code, not vice versa
- Possibility of program verification using Model Checking
- You can check temporal properties (LTL)
Issues

• Hard to build an FSM with desired structure and behavior
• One of approaches – mutation-based metaheuristics
Mutation-based FSM learning

- Evolution Strategies (ES)
- Genetic Algorithms (GA)
- Mutation-based Ant Colony Optimization (MuACO)

All these algorithms use FSM mutations
Mutation-Based Ant Colony Optimization

• Proposed by the authors of this work in 2012
• Based on Ant Colony Optimization
• Can be thought of as an ES “with memory”
• No time to go into detail 😞
Solution representation

<table>
<thead>
<tr>
<th>Transition table</th>
<th>Output table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>δ</strong></td>
<td><strong>Event</strong></td>
</tr>
<tr>
<td><strong>State</strong></td>
<td><strong>A</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Learning FSMs: Conserving Fitness Evaluations
 FSM Mutations

Change transition action

Change transition end state

Learning FSMs: Conserving Fitness Evaluations
Proposed idea
Fitness evaluation: used transitions

1. Calculated fitness
2. Used transitions

Learning FSMs: Conserving Fitness Evaluations
Mutation

Mutate this transition

Learning FSMs: Conserving Fitness Evaluations
Observation

Mutation

- Used transition
- Unused transition

Compute fitness

Fitness has not changed
Challenge

OK, found a way not to calculate fitness of FSMs in some cases

1. Can we design an efficient implementation?
2. Will it make a difference in performance?
3. Limits of applicability?
Implementation

• Store an array of transition usage marks for each FSM
• Mark used transitions during fitness evaluation
• Copy marks when not calculating fitness
Domain knowledge

• Black box – no domain knowledge
• General domain knowledge about FSMs
• Problem-specific domain knowledge
Experiments
Experiments: Purpose

• Does it make a difference?
• How much resources does it require?
Experiments: Algorithms

- Evolutionary strategy (ES)
- Genetic algorithm (GA)
- Mutation-Based Ant Colony Optimization for learning FSMs (MuACOsm)
Experiments: Problems

1. Artificial Ant Problem
2. Learning EFSMs from tests
General experimental setup

1. Tune each algorithm for time $t_{\text{tune}}$
   - Using full factorial design of experiment
2. Run each algorithm with tuned parameters
Artificial Ant Problem

- Toroidal field $N \times N$
- $M$ pieces of food
- $K$ time steps
- Fixed position of food and the ant
- Goal – build an FSM, such that the ant will eat all food in $K$ steps

Field example: John Muir Trail
Artificial Ant: Fitness function

\[ f = n_{\text{food}} + \frac{K - s_{\text{last}} - 1}{K} \]

- \( n_{\text{food}} \) – number of eaten food pieces
- \( K \) – max number of allotted steps
- \( s_{\text{last}} \) – number of used steps

Learning FSMs: Conserving Fitness Evaluations
Success rate

- Successful run: fitness $\geq 89$
- Success rate = $\frac{N_{\text{successful runs}}}{N_{\text{runs}}}$
Experiment design

- Vary number of states
- Limited number of fitness evaluations
- Measure:
  - Success rate
  - Time
ES median time

- **plain**
- **with marking**

- **Median time, sec.**
  - 0.6
  - 0.5
  - 0.4
  - 0.3
  - 0.2

- **Number of FSM states**
  - 4
  - 6
  - 8
  - 10
  - 12
  - 14
  - 16
  - 18
  - 20

Graph showing the median time for ES with and without marking as a function of the number of FSM states.
GA median time

- **plain**
- **with marking**

Median time, sec.

Number of FSM states
MuACO median time

![Graph showing the median time for plain and with marking methods against the number of FSM states.](image)
MuACO Fitness

Mean fitness vs Number of FSM states for Plain and With marking.
Fitness evaluation time: plain algorithms

![Graph showing fitness evaluation time for different algorithms.](image)

- **es**: Blue line
- **ga**: Green line
- **muaco**: Red line

The graph plots fitness evaluation time as a percentage of total time against the number of FSM states. The lines show how each algorithm's performance changes with varying numbers of states.
Fitness evaluation time: with marking

![Graph showing fitness evaluation time with marking]
Statistical Significance

- ANOVA test
- Fitness distributions significantly different for ES and MuACOsm
- Insignificant for GA
Learning Extended Finite-State Machines from tests (1)
Learning Extended Finite-State Machines from tests (2)

Input data:

- Number of states $C$ and sets $\Sigma$ and $\Delta$
- Set of test examples $T$
- $T_i = \langle$ input sequence $I_j$, output sequence $O_j \rangle$

NP-hard problem: build an EFSM with $C$ states compliant with tests $T$
Example of a test

A H T[x₁] T[x₁ & x₂] → z₁ z₁ z₂ z₅ z₇
Learning EFSMs: Fitness function

- Pass inputs to EFSM, record outputs
- Compare generated outputs with references
- Fitness = string similarity measure (edit distance)

\[
f' = \frac{1}{|T|} \sum_{j=1}^{\mid T \mid} \left( 1 - \frac{ED(O_j, A_j)}{\max(len(O_j), len(A_j))} \right)
\]

\[
f = 100 \cdot f' + \frac{1}{100} \cdot (100 - n_{\text{trans}})
\]
Experimental setup

1. Generate random EFSM with $C$ states
2. Generate set of tests of total length $C \times 150$
3. Learn EFSM from tests
4. Experiment for each $C$ repeated 100 times
5. Limited number of fitness evaluations
Success rate

Number of EFSM states

Success rate, %

plain
with marking
Median time vs. Number of EFSM states
Time for fitness evaluation

![Graph showing fitness evaluation time as a percentage of total time against the number of FSM states. The graph includes two lines, one for plain and one for enhanced, showing the increase in efficiency as the number of states increases.]
Conclusion

Developed approach

– Applicable to all FSM learning algorithms that use mutations
– Allows to explore more FSMs with the same number of fitness evaluations
– Effectively improves fitness and time
Limitations

• Makes sense to use if cost of fitness computation is relatively high
Future work

Explore more ways of using domain knowledge in FSM learning algorithms
Acknowledgements

• University ITMO research project 610455
Thank you for your attention!

Learning FSMs: Conserving Fitness Evaluations by Marking Used Transitions

Daniil Chivilikhin
Vladimir Ulyantsev

{chivdan,ulyantsev}@rain.ifmo.ru

This presentation online: