An Evolutionary Methodology for Automatic Design of Finite State Machines*

J. Manuel Colmenar†, Alfredo Cuesta-Infante†, José L. Risco-Martín⋆, J. Ignacio Hidalgo⋆
† C.E.S. Felipe II, Complutense University of Madrid, 28300 Aranjuez, Spain
{jmcolmenar, alfredo.cuesta}@ajz.ucm.es
⋆ Dept. of Computer Architecture and Automation, Complutense University of Madrid, 28040 Madrid, Spain
{jlrisco, hidalgo}@dacya.ucm.es

ABSTRACT

We propose an evolutionary flow for finite state machine inference through the cooperation of grammatical evolution and a genetic algorithm. This coevolution has two main advantages. First, a high-level description of the target problem is accepted by the flow, being easier and affordable for system designers. Second, the designer does not need to define a training set of input values because it is automatically generated by the genetic algorithm at run time. Our experiments on the sequence recognizer and the vending machine problems obtained the FSM solution in 99.96% and 100% of the optimization runs, respectively.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program synthesis; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods

General Terms

Algorithms

Keywords

Grammatical evolution; Genetic programming; Genetic algorithms; Design/synthesis; Finite state machines

1. INTRODUCTION

Finite state machines (FSMs) have been commonly used to describe many kind of state-based systems like, for instance, cluster labeling, fault detection, pattern recognition or hardware design and verification. It is well known that they provide a high-level mechanism to govern a complex system and, starting from an FSM, many design tools are able to generate its hardware system.

Many previous works like [2, 5], tried to solve the inference of FSMs. But they usually require a given training set of input values and/or rely on complex formulae, rules or constraints to describe the problem at hand.

*This work has been supported by Spanish Government grants TIN2008-00508 and MEC Consolider Ingenio CSD00C-07-20811 of the Spanish Council of Science and Technology.

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
ACM 978-1-4503-1964-5/13/07.

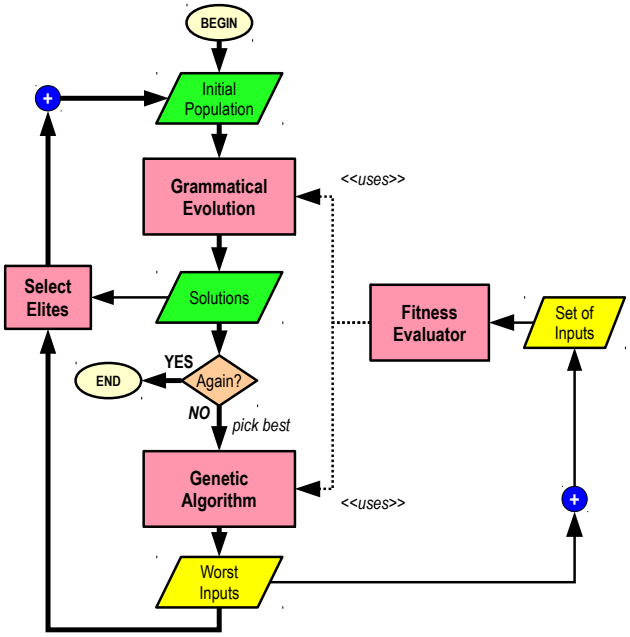

Figure 1: Optimization flow. Thick arrows represent the execution path, regular arrows indicate data transmission and dotted arrows indicate usage.

In this paper we propose an optimization flow that, by means of grammatical evolution (GE) in collaboration with a genetic algorithm (GA), is able to find the FSM description that solves a computational problem and verifies an automatically generated set of input values.

2. EVOLUTIONARY FLOW

We have chosen *Grammatical Evolution (GE)* [4] as the technique to explore and limit the search space which, in our proposal, is formed by FSMs.

The evolutionary scheme we propose is shown in Figure 1. The Grammatical Evolution (GE) module receives an initial population of FSMs randomly generated. Notice that, in its execution, GE uses the Fitness Evaluator (FIT) module to test each candidate FSM. The fitness function is a measure of distance, in number of different output values, between the output generated by the FSM under evaluation and the expected (correct) output.

The expected output is generated by a high-level code

Table 1: Results for 3-bits pattern recognizer.

Pattern	Golden	Reducible	NO sol.
000	23.33 %	76.66 %	0
001	65.52 %	34.48 %	0
010	93.10 %	6.89 %	0
011	93.10 %	3.45 %	1
100	92.86 %	7.14 %	0
101	96.55 %	3.45 %	0
110	63.33 %	36.66 %	0
111	23.33 %	76.66 %	0
Total	68.68 %	31.28 %	0.04 %

(Java) that solves the problem. The creation of this code is usually easier than the obtention of the reduced FSM. Therefore, our proposal can be applied to any problem that could be described through a high-level code.

Once GE finishes, and before reaching the maximum number of iterations, the Genetic Algorithm (GA) module executes over the best solution coming from GE.

GA module tries to find out the input string that maximizes the number of failures for each given FSM, and incorporates the resulting strings to the set of inputs that FIT processes. Therefore, a feedback between GA and FIT is produced, which means that, in the next iteration, the FSMs processed by GE will be tested over a harder objective function, automatically generating the training set.

As seen in Figure 1, the best solutions (elites) coming from the previous execution of GE are inserted as immigrants into the GE new population.

We have implemented our optimization flow in Java. The GE module was coded using GEVA [3], while the rest of the code was developed by us. The algorithms, programs and tests are publicly available in our Java Evolutionary Computation library (JECO) at Google Code [1].

3. EXPERIMENTAL RESULTS

We have tested our proposal with two problems: the 3-bit overlapped sequence recognizer and the vending machine problem. In both of them we use input strings of 100 bits length, which corresponds to a huge search space for the input set (2^{100} combinations).

In the sequence recognizer problem we run 30 optimizations iterating 50 times the GE main loop for each one of the eight 3-bits different patterns. After each run, we obtained one FSM as a solution. Table 1 shows the results.

As seen, our algorithm obtained the reduced FSM solution (called "Golden") in 68.68% of the runs and an equivalent reducible solution in 31.28% of the runs. Just one run for the 011 pattern did not find any solution. Notice that partially correct FSMs are not considered as solutions.

The vending machine problem represents a different scenario: (1) the machine dispenses a product after receiving a certain amount of money or more; (2) after dispensing the product (output will be 1) the machine is reset to the initial state providing no change; (3) the machine accepts two kind of coins: 5¢, and 10¢.

In the case that the amount to reach was 15¢, five different paths lead to five different final states, as represented in the FSM shown in Figure 2 (a), having a number of nine states. However, this FSM can be reduced to a simpler machine having just four states, as shown in Figure 2 (b). Therefore,

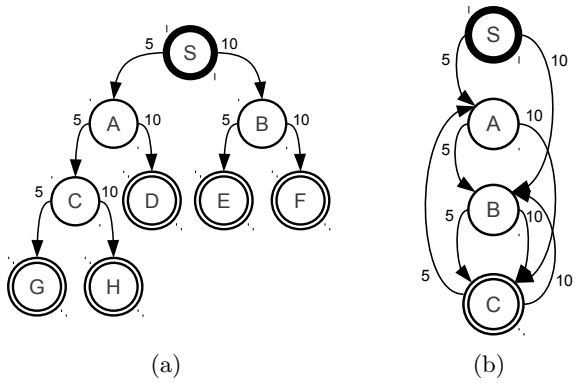

Figure 2: (a) FSM solving 15¢ vending machine. Transitions from the final states to S were removed. (b) Reduced FSM for the 15¢ vending machine.

finding a reduced FSM that solve the problem is harder than in the sequence recognizer problem.

We tackled the vending problem for an amount of 15¢ with the same experimental setup as in the pattern recognition problem. The optimization was successful because all the 30 runs obtained the golden FSM, similar to Figure 2 (b).

4. CONCLUSIONS AND FUTURE WORK

In this paper, we propose an evolutionary flow where FSM implementations are automatically obtained from scratch through a grammatical evolution process. Using a genetic algorithm, the flow automatically generates the training set of inputs that will be tested by the fitness function, finding out those inputs with the highest failure rate. In addition, no complex rules or formula are required. On the contrary, a Java code solving the tackled problem is needed by the fitness module.

Future work involves parallelization of the evaluation phase in order to tackle more complex problems.

5. REFERENCES

[1] JECO: Java Evolutionary Computation library. `https://code.google.com/p/paba/`.

[2] C. Johnson. Genetic programming with fitness based on model checking. In M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. Esparcia-Alcázar, editors, *Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124. Springer Berlin / Heidelberg, 2007.

[3] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. GEVA - grammatical evolution in Java. *SIGEVOlution*, 3(2):17–22, 2008.

[4] M. O. C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[5] F. Tsarev and K. Egorov. Finite state machine induction using genetic algorithm based on testing and model checking. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 759–762, New York, NY, USA, 2011. ACM.