Inferring Temporal Properties of Finite-State Machine Models with Genetic Programming

Daniil Chivilikhin
ITMO University
49 Kronverkskiy av.
Saint Petersburg, Russia
chivdan@rain.ifmo.ru

Ilya Ivanov
ITMO University
49 Kronverkskiy av.
Saint Petersburg, Russia
ilya.ivanov.ifmo@gmail.com

Anatoly Shalyto
ITMO University
49 Kronverkskiy av.
Saint Petersburg, Russia
shalyto@mail.ifmo.ru

ABSTRACT

The paper presents a genetic programming based approach for inferring general form Linear Temporal Logic properties of finite-state machine models. Candidate properties are evaluated using several fitness functions, therefore multiobjective evolutionary algorithms are used. The feasibility of the approach is demonstrated by two examples.

CCS Concepts

•**Software and its engineering** → **Search-based software engineering;**

Keywords

Design/Synthesis, Software Engineering, Empirical study

1. INTRODUCTION

Finite-state models are often used to represent software behavior and can be employed for various automated verification and testing procedures. They also provide a quick picture of how a piece of software works. However, such models are rarely kept up to date with a software project, even if they have been created in the first place. That is why a large number of model inference techniques have been proposed [2, 7, 5, 4]. The general approach is to record execution traces of a piece of software and try to generalize them into some model.

Another useful thing to have in a software project is its formal specification expressed in temporal logics, e.g. Linear Temporal Logic (LTL). Such specifications are often written for finite-state machine (FSM) models of software. Having a model and complimentary temporal properties is essential for formal verification. Several approaches for mining temporal specifications have been proposed, e.g. Perracotta [10], which works with execution traces and tries to generalize them into some temporal logic formulas. A common drawback of such techniques is that they only seem to be capable of inferring very simple formulas.

GECCO '15, July 11–15, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739482.2768475

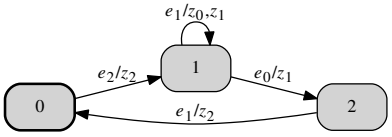

Figure 1: An example of a finite-state machine

In this paper we propose to solve the problem of temporal specification inference in the assumption that the FSM model is already known. The model can be human-written or inferred by the aforementioned automated techniques. The contribution of the paper is an approach based on genetic programming (GP) [6] for evolving temporal logic formulas that: (1) hold for the given FSM, (2) are not too long or too complex and (3) are "interesting". An example of an uninteresting formula is a tautology. Since these search objectives are conflicting by nature, multiobjective optimization is used.

2. PROBLEM STATEMENT

In this section we provide a more formal problem statement. We first briefly review the main concepts used in this study.

The goal of this research is finding temporal properties of a given FSM. An FSM is a six-tuple $\langle E, Y, Z, y_0, \phi, \delta \rangle$, where E is a set of input events, Y is a set of states, $y_0 \in Y$ is the initial state, Z is a set of output actions, $\phi\colon Y \times E \to Y$ is the transitions function and $\delta\colon Y \times E \to Z^*$ is the outputs function. An example of an FSM is given in Fig. 1: each transition is marked with an event (before the slash) and a series of output actions (after the slash). The initial state is marked bold.

The LTL language consists of problem dependent propositional variables, Boolean logic operators $\vee$, $\wedge$, $\neg$, $\rightarrow$, and the following set of temporal operators. ***G**lobally* – $G(f)$ means that f has to hold for all states. *ne**X**t* – $X(f)$ means that f has to hold in the next state. ***F**uture* – $F(f)$ means that f has to hold in some state in the future. ***U**ntil* – $U(f, g)$ means that f has to hold until g holds. ***R**elease* – $R(f, g)$ means that g has to hold until f holds, or g has to fold forever if f never holds.

In this work we use the following propositional variables: wasEvent$(e), e \in E$ (a transition marked with input event $e \in E$ has been triggered) and wasAction$(z), z \in Z$ (a tran-

sition marked with output action $z \in Z$ has been triggered). To check whether an LTL formula holds for an FSM, model checkers (e.g. *SPIN*[1]) are used. For example, formula

$$G\left(\text{wasEvent}(e_2) \rightarrow \text{wasAction}(z_2)\right),$$

which means that emitting event e_2 in some state always causes emitting action z_2, holds for the FSM from Fig. 1. On the other hand, formula

$$G\left(\text{wasEvent}(e_1) \rightarrow \text{wasAction}(z_2)\right)$$

does not hold, since there is a transition in the FSM that is marked with e_1 but not with z_2.

Let a be the given FSM. The goal is to find a set of LTL formulas $\{f_a\}$ satisfying the following constraints.

1. All formulas from $\{f_a\}$ must hold for FSM a.
2. Formulas should not be too structurally complex: we are more interested in shorter formulas than in longer ones.
3. Formulas should be "interesting": we are interested in formulas that hold for the input FSM and do not hold for all other FSMs.

The third constraint is rather informal. We found that no single fitness function (FF) is sufficient to express it, so several functions were used simultaneously.

3. PROPOSED APPROACH

The general idea of the proposed approach is to use GP to evolve a population of LTL formulas represented in tree form. We use a standard implementation of GP from the evolutionary algorithms library *ECJ*[2] with Koza's standard tree mutation and crossover operators; NSGA-II [3] and SPEA2 [11] are used as multiobjective algorithms. In this research we focus on formulas of the form $G(f)$, where f is an LTL formula without a single "G" operator.

Below we describe the objective FFs we propose for LTL formula inference. The first objective function F_1 measures the degree to which the input FSM satisfies a candidate LTL formula. Function F_2 is used for minimizing formula complexity, functions F_3–F_4 are used for enforcing interesting formulas.

3.1 Formula Must Hold for Input FSM

This is the main search objective – we are only interested in formulas that hold for the input FSM a. To determine whether an LTL formula holds for the input FSM, the FSM verifier (model checker) developed in [8] is used.

Normally, a verifier returns *true* if a model satisfies a temporal formula and *false*, otherwise. Obviously, an FF with only two possible values would provide no gradient to the search. A distinctive feature of the aforementioned FSM verifier is that it returns the number of FSM transitions that certainly do not belong to the counterexample. Such transitions are called *verified*. Using this feature, the first objective function F_1 is defined as the ratio of the number of verified transitions to the total number of reachable transitions:

$$F_1(f) = r(a, f) = \frac{n_{\text{verified}}(a, f)}{n_{\text{transitions}}(a)},$$

where a is the FSM, f is the LTL formula, $n_{\text{verified}}(a, f)$ is the number of verified transitions in a with respect to f, and $n_{\text{transitions}}(a)$ is the total number of reachable transitions in a. The values of $r(a, f)$ lie in the interval $[0, 1]$ and $r(a, f) = 1$ iff formula f holds for FSM a. Note that though non-perfect formulas are allowed during search, a formula has to hold for the FSM in order to be included in the final set of solutions.

3.2 Minimal Formula Weight

We are interested in formulas that are not too structurally complex. To measure the structural complexity of a formula, the notion of *formula weight* is used. Suppose $O = \{\vee, \wedge, \neg, \rightarrow, X, F, U, R\}$ is the set of operators and S is the set of propositional variables: $S = \bigcup_{e \in E} \text{wasEvent}(e) \cup \bigcup_{z \in Z} \text{wasAction}(z)$. We assign each operator $o \in O$ and propositional variable $s \in S$ their weight, w_o and w_s, respectively. Formula weight W is defined in the following way:

$$W(s) = w_s, s \in S;$$

$$W(o(\text{arg}_1, \ldots, \text{arg}_n)) = w_o + \sum_{i=1}^{n} W(\text{arg}_i), n = \{1, 2\}.$$

Tthe second objective function F_2 is defined as: $F_2(f) = \frac{1}{W(f)}$. If operator and propositional variable weights are greater than or equal to one, then the values of F_2 lie in the interval $[0, 1]$ and increase with the decrease of formula weight.

3.3 Formula Should not Hold for Randomly Generate FSMs

Let A_f be the set of all FSMs that satisfy formula f. The power of this set $|A_f|$ can be viewed as a notion of formula specificity – we are interested in formulas that hold for the particular given FSM. Roughly speaking, the smaller $|A_f|$, the more information f bears. However, fully computing A_f is infeasible, so an approximation is used. For calculating the third objective function F_3, we first generate N_{sample} random FSMs $a_1, \ldots, a_{N_{\text{sample}}}$ with the same number of states, sets of input events, input variables, and output actions as the input FSM a. For each of the generated FSMs a_i we run the FSM verifier and get the corresponding value $r(a_i, f)$. Then the third objective function is defined as:

$$F_3(f) = \frac{1}{1 + \sum_{i=1}^{N_{\text{sample}}} r(a_i, f)^2}.$$

The smaller the number of random FSMs that satisfy the candidate formula, the greater the value of F_3.

3.4 Formula Should not Hold for Mutants of the Input FSM

This objective function serves as another approximation of $|A_f|$. A mutant of an FSM is another FSM that slightly differs from the original one. A set of mutants $m_1, \ldots, m_{N_{\text{sample}}}$ of the input FSM is generated using the following two mutation operators.

Change transition end state. For a randomly selected transition, the state y it leads to is changed to another state selected uniformly at random from $Y \setminus \{y\}$.

Add or delete transitions. The set of transitions in each state is modified with a certrain probability. In case of adding a transition, the new transition is added that leads to

[1]http://spinroot.com/

[2]http://cs.gmu.edu/eclab/projects/ecj

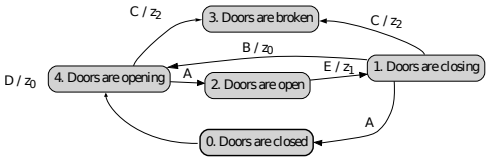

Figure 2: Elevator doors control FSM

a state selected uniformly at random from Y and is marked with a random event and output actions sequence. In case of deleting a transition, a randomly selected transition is deleted.

Each time the mutation operator to apply is selected uniformly at random. Objective function F_4 is calculated as:

$$F_4(f) = \frac{1}{1 + \sum_{i=1}^{N_{\text{sample}}} r(m_i, f)^2}.$$

The smaller the number of mutants that satisfy the candidate formula, the greater the value of F_4.

The motivation behind this FF is that even small changes to the model should be sufficient to violate its important properties. That is why we are interested in steering the search towards formulas that are violated by such small changes to the model.

3.5 Formula Should not Hold for FSM Constructed from Scenarios

A *scenario* is a finite path in an FSM that starts from the initial state. For example, the following is a scenario derived from the FSM in Fig. 1:

$$\langle e_2, (z_2)\rangle, \langle e_1, (z_0, z_1)\rangle, \langle e_0, (z_1)\rangle.$$

Here we generate a set of scenarios from the input FSM. Then we use the algorithm from [9] to build an FSM a^* that is compliant with all scenarios.

The main idea behind this objective function is that since scenarios length is limited, not all formulas that hold for a will hold for the derived FSM a^*. This way we try to focus the search on more general formulas. Objective function F_5 is calculated as $F_5(f) = 1 - r(a^*, f)$.

3.6 Formula should not Hold for Mutants of the FSM Constructed from Scenarios

This objective function is analogous to F_4 except we generate mutants of the FSM constructed from scenarios, rather than mutants of the input FSM.

4. EXPERIMENTS AND RESULTS

In this section we describe the experiments we performed and the subsequent results.

4.1 Case Study: Elevator Doors Control FSM

For experimentation we chose the elevator doors control FSM from [8] shown in Fig. 2. The initial state is state number zero. The FSM has five input events: A (doors are fully opened or fully closed), B (an obstacle prevents doors from closing), C (doors are broken), D ("Open doors" button has been pressed), E ("Close doors" button has been pressed). There are three output actions: z_0 (start opening doors), z_1 (start closing doors), z_2 (call the emergency).

The doors are initially closed. When the "Open doors" button is pressed, the doors start opening. If all goes well, doors are opened. If doors are broken, the emergency is called. If the doors are open and the "Close doors" button is pressed, the doors start closing, unless they are broken (and the emergency is called). If an obstacle prevents doors from closing, they start opening again. The FSM complies with a set of 17 LTL properties, e.g.:

$$G\left(\text{wasEvent}(D) \rightarrow \text{wasAction}(z_0)\right).$$

4.2 Assessing LTL Formula Quality

One important question is how to assess the quality of inferred LTL formulas, since no obvious formal criterion is present. We use two empirical metrics described below, both are meant to approximate how many formulas from the original specification were discovered by the proposed approach. Let $\{f_{\text{old}}\}$ be the original set of LTL formulas and $\{f_{\text{new}}\}$ be the set of LTL formulas inferred by the algorithm.

Coverage metric. The algorithm from [1] is used to learn an FSM a' from the set of scenarios (see Section 3.5) and the set of inferred formulas $\{f_{\text{new}}\}$. Then, we check how many of the formulas from the original specification $\{f_{\text{old}}\}$ does the constructed FSM satisfy:

$$c_{\text{cover}} = \frac{\sum_{f \in \{f_{\text{old}}\}} r(a', f)}{|\{f_{\text{old}}\}|}.$$

The larger this value, the more information about the original FSM the inferred formulas bear.

Mutants metric. To calculate this metric we first generate $M' = 1000$ different mutants $m_1, \ldots, m_{M'}$ of the input FSM a. Then we calculate the ratio of mutants that violate *at least one* formula from $\{f_{\text{old}}\}$:

$$n^{\text{old}}_{\text{unsat}} = \frac{1}{M'} \sum_{i=1}^{M'} \left(1 - \min_{f \in \{f_{\text{old}}\}} \lfloor r(m_i, f) \rfloor\right).$$

Next, we use the same mutants to calculate $n^{\text{new}}_{\text{unsat}}$ – the ratio of mutants that violate at least one formula from the inferred set $\{f_{\text{new}}\}$. Finally, the metric is calculated as the ratio:

$$c_{\text{mut}} = \frac{n^{\text{new}}_{\text{unsat}}}{n^{\text{old}}_{\text{unsat}}}.$$

4.3 Experimental Setup and Results

In experiments we tested different combinations of the described FFs. The first two FFs are essential and were used in all runs, other functions were tested in all possible combinations.

The algorithm was run for 50 generations with a population size of 500. Both SPEA2 and NSGA-II were run 20 times with each configuration. The size of the elite archive of SPEA2 was fixed at 100. The result of each run is the set of LTL formulas in the Pareto front constructed from Pareto fronts of all generations.

Experimental results are presented in Table 1. In the last three columns the value before the slash is for SPEA2 and the value after the slash is for NSGA-II. Below are a few examples of generated formulas:

$G\left(F\left(\text{wasAction}(z_1)\right) \rightarrow \neg\, \text{wasAction}(z_2)\right)$

$G\left(\text{wasAction}(z_0) \rightarrow F\left(\text{wasAction}(z_1) \vee X\left(\text{wasAction}(z_2)\right)\right)\right)$

Table 1: Experimental configurations, median values of c_{cover} and c_{mut} metrics, and run times for SPEA2/NSGA-II

№	F_3	F_4	F_5	F_6	$100 \cdot c_{\text{cover}}$, %	$100 \cdot c_{\text{mut}}$, %	**Time, s.**
1	-	-	-	-	44.1 / 44.1	53.4 / 38.5	60 / 14
2	-	-	-	+	64.7 / 58.8	49.6 / 36.6	170 / 78
3	-	-	+	-	73.5 / 70.6	65.3 / 58.0	133 / 84
4	-	-	+	+	88.2 / 88.2	77.5 / 83.6	521 / 2493
5	-	+	-	-	58.8 / 58.8	55.3 / 49.2	152 / 159
6	-	+	-	+	73.5 / 79.4	71.0 / 74.0	889 / 2898
7	-	+	+	-	88.2 / 79.4	78.6 / 79.4	579 / 2197
8	-	+	+	+	88.2 / **88.2**	83.2 / **86.4**	1894 / 4618
9	+	-	-	-	53.0 / 61.8	42.4 / 42.0	64 / 17
10	+	-	-	+	67.6 / 64.7	44.7 / 46.6	158 / 108
11	+	-	+	-	88.2 / 82.4	71.4 / 69.5	141 / 211
12	+	-	+	+	88.2 / 88.2	77.5 / 80.9	632 / 2025
13	+	+	-	-	67.6 / 58.8	66.4 / 56.9	236 / 195
14	+	+	-	+	64.7 / 79.4	71.0 / 69.1	796 / 2259
15	+	+	+	-	**88.2** / 88.2	**87.8** / 85.5	876 / 1775
16	+	+	+	+	88.2 / 82.4	84.0 / 83.6	1618 / 4724

Experimental data in Table 1 suggests that SPEA2 and NSGA-II yield similar performance, however in most cases SPEA2 is much faster than NSGA-II. For this reason we used SPEA2 in all remaining experiments.

Median values of metrics indicate that the best configuration for SPEA2 is № 15 and for NSGA-II the best is № 8 (best values are highlighted). We used the Wilcoxon signed-rank test to verify that. According to the test with respect to c_{cover} metric, configuration № 15 is statistically better (p-value of accepting alternative hypothesis is less than 0.05) than all except 4, 7, 8, and 12. According to the c_{mut} metric, № 15 is in a tie with 7, 8, and 16, and better than all others.

To further investigate the behavior of the proposed approach we performed additional experiments with configuration № 15 and SPEA2 algorithm. We varied the population size from 100 to 1000 (archive size was kept at 20% of population size), which resulted in $100 \cdot c_{\text{learn}}$ of 23%, 86%, 86%, 86% and $100 \cdot c_{\text{mut}}$ of 13%, 79%, 96%, 96% for population sizes 100, 250, 500, and 1000, respectively. Furthermore, changing the number of generations from 25 to 200 does not significantly influence the quality of generated formulas.

Finally, we used our approach to infer LTL properties of an FSM for controlling an ATM machine which has 12 states, 14 events, 13 output actions and has a human-written set of 30 LTL formulas. Our algorithm was able to reach a value of $100 \cdot c_{\text{mut}} = 65\%$. Calculating the coverage metric in this case is infeasible due to the large dimensionality of the problem.

5. CONCLUSIONS AND FUTURE WORK

The most closely related research is Perracotta [10], a dynamic technique for inferring simple temporal properties. Temporal property inference is based on a hierarchy of predefined property templates, which are simple regular expressions. The program is instrumented and run through a set of tests, resulting in a set of execution traces. The algorithm infers the strictest possible template two events satisfy. Authors also deal with the issue of imperfect traces, allowing them to apply their technique for large programs, such as the Windows kernel. The main differences of our work from Perracotta are that we, first, work with models rather than execution traces, and, second, aim to infer general form temporal formulas.

We have proposed an approach for inferring general form LTL properties of FSM models using GP and multiobjective evolutionary algorithms. Experiments showed that, according to two introduced performance metrics, the approach was able to infer up to 100% of human-written LTL formulas of the studied FSMs. Future work includes applying the proposed approach to find temporal properties of FSM models inferred by existing model inference techniques.

This work was financially supported by the Government of Russian Federation, Grant 074-U01.

6. REFERENCES

[1] D. Chivilikhin and V. Ulyantsev. MuACOsm: A New Mutation-Based Ant Colony Optimization Algorithm for Learning Finite-State Machines. In *Proceedings of the fifteenth annual conference on Genetic and evolutionary computation*, GECCO '13, pages 511–518, New York, NY, USA, 2013. ACM.

[2] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining Object Behavior with ADABU. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 17–24, New York, NY, USA, 2006. ACM.

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

[4] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[5] M. Heule and S. Verwer. Software model synthesis using satisfiability solvers. *Empirical Softw. Engg.*, 18(4):825–856, Aug. 2013.

[6] J. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, 1992. Cambridge, MA, USA.

[7] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, pages 501–510. ACM, 2008.

[8] 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.

[9] V. Ulyantsev and F. Tsarev. Extended Finite-State Machine Induction Using SAT-Solver. In *Proceedings of the Fourth International Conference on Machine Learning and Applications*, volume 2, pages 346–349, Los Alamitos, CA, USA, 2011.

[10] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291. ACM, 2006.

[11] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm, 2001.