Formal Verification of 800 Genetically Constructed Automata Programs: A Case Study

Mikhail Lukin, Maxim Buzdalov, and Anatoly Shalyto

ITMO University
49 Kronverkskiy prosp.
Saint-Petersburg, Russia, 197101
{lukinma,mbuzdalov}@gmail.com, shalyto@mail.ifmo.ru

Abstract. Engineering of mission critical software requires a program to be verified that it satisfies a number of properties. This is often done using model checking. However, construction of a program model to be verified and analyzing counterexamples is not an easy task. This can be made easier with the automata-based programming paradigm.

There exist some cases when there are many programs to verify and it is impossible to construct a precise enough finite-state model of the environment. We present an approach for automata program verification under such conditions. Our case study is based on 800 automata programs which solve a simple path-planning problem. As a result, we verified that at least 231 of them are provably correct.

Keywords: automata-based programming, formal verification, model checking.

1 Introduction

Engineering of mission critical software requires a program to be verified that it satisfies a number of properties. This is often done using the model checking approach [7]. However, construction of a program model to be verified and analysing counterexamples is not an easy task.

Automata-based programming [3, 8, 9] is a programming paradigm which proposes to design and implement software systems as systems of interacting automated controlled objects. Each automated controlled object consists of a controlling extended finite-state machine (EFSM) and a controlled object. One of the main advantages of automata-based programming is that automata programs can be effectively verified using the model checking approach. Automata programs are isomorphic to their own models, which automates many steps needed to verify an automata program [5, 10]. This makes automata-based programming a good tool in industry, a notable example of which is a new standard for distributed control and automation IEC 61499 [11].

In some cases, synthesis of automata programs is possible using search-based software engineering methods, such as genetic algorithms [1, 2, 10]. This may lead to existence of many programs to check, and their underlying logic can be

E. Yahav (Ed.): HVC 2014, LNCS 8855, pp. 165–170, 2014.

cumbersome. What is more, it is sometimes impossible to construct a finite-state model of the environment that is enough to verify the necessary properties.

In this paper, an approach for automata program verification, which can cope with such conditions, is presented. We illustrate this approach on a case study, which is based on 800 automata programs constructed by a genetic algorithm. We verified that 231 of them are provably correct.

2 Problem Formulation

In the paper [1] solutions to the path-planning problem were constructed in the form of finite state machines using a genetic algorithm. More precisely, the path planning problem with incomplete information was addressed: an *agent* with $O(1)$ memory and only contact sensors has to find a *target* in an unknown area with finite obstacles. In the paper [6] some algorithms (the most known are BUG-1 and BUG-2) are given which find the target or determine that the target is unreachable in finite number of steps.

The paper [1] considered a discretized version of this problem, which is more suitable to experiments with automata program synthesis. The field is an infinite square grid. Each grid cell is either free or contains an obstacle. Each eight-connected group of cells with obstacles has a finite size. One of the cells without an obstacle is declared to be a target. An agent occupies an entire cell. The position of the agent is determined by its Cartesian coordinates and direction (N, W, S or E). The next cell in this direction is said to be the *adjacent* cell. The agent has $O(1)$ additional memory which is used to store a single position of the agent. The agent's logic is encoded as an EFSM.

The agent has access to the following data: (X_t, Y_t) – the target location, (X_a, Y_a, D_a) – the agent's coordinates and direction, (X_s, Y_s, D_s) – saved coordinates and direction, (X_j, Y_j) – coordinates of the adjacent cell (which is a function of X_a, Y_a, D_a), O – is there an obstacle in the adjacent cell. These data are converted to the Boolean variables which the agent directly accesses:

- "can move forward": `x1` = `not` O;
- "is move forward cool": `x2` = $dist(X_j, Y_j, X_t, Y_t) < dist(X_a, Y_a, X_t, Y_t)$;
- "is at finish": `x3` = $X_a = X_t$ `and` $Y_a = Y_t$;
- "is at saved": `x4` = $X_a = X_s$ `and` $Y_a = Y_s$ `and` $D_a = D_s$;
- "is better than saved": `x5` = $dist(X_a, Y_a, X_t, Y_t) < dist(X_s, Y_s, X_t, Y_t)$;

where $dist(X_1, Y_1, X_2, Y_2) = |X_1 - X_2| + |Y_1 - Y_2|$. The possible actions are:

- "move forward": move forward to the adjacent cell;
- "rotate positive": rotate 90 degrees clockwise;
- "rotate negative": rotate 90 degrees counter-clockwise;
- "report reached": terminate and say it has reached the target;
- "report unreachable": terminate and say the target is unreachable;
- "save position": save current coordinates and direction to memory;
- "do nothing": do nothing.

The agent may end up in one of the following ways:

1. It moves to a cell which contains an obstacle ("crashes").
2. It enters a loop in the state space.
3. It moves apart from the target forever.
4. It performs the "report reached" action and is **not** located at the target.
5. It performs the "report reached" action and is located at the target.
6. It performs the "report unreachable" action, the target is **not** unreachable.
7. It performs the "report unreachable" action, the target is unreachable and the agent has **not** visited all cells that are surrounding the eight-connected obstacle component which contains the target inside.
8. It performs the "report unreachable" action, the target is unreachable and the agent has visited all cells mentioned in the previous case.

From these ways, only the cases 5 and 8 refer to the correct termination.

In the paper [1], it was reported that 800 EFSMs were evolved using genetic programming and some rudimentary coevolution with tests. They were extensively tested and have never failed. However, a formal proof for their correctness is missing in [1].

3 Proposed Verification Approach

The correctness of an agent can follow from two statements only: "for any field with a reachable target, the agent will eventually reach the target and perform the "report reached" action" and "for any field with an unreachable target, the agent will eventually perform the "report unreachable" action". If a field were fixed, checking these statements would be possible by a partial breadth-first traversal of a graph, whose vertices are possible program states. However, the fields are not fixed, which makes the statements inexpressible in terms of states or paths in the agent's EFSM.

We suggest the following workflow:

1. Construct a hypothesis of how a series of EFSMs work.
2. Construct a (probably lossy and non-deterministic) finite-state model of an agent and the environment, and a set of LTL formulae that together can be used by a model checker to prove that a given EFSM satisfies the hypothesis.
3. Prove formally that any EFSM that satisfies the hypothesis is correct.
4. Run a model checker on available EFSMs using the model and formulae from step 2. All EFSMs that are successfully verified are correct.

In Section 4 we apply this workflow, step by step, to verification of 800 finite-state machines solving the path planning problem described in Section 2.

4 Application to the Path Planning Problem

After preliminary experiments with several agents we hypothesized that they follow the BUG-2 scheme scheme [6]. Such agents move towards the target while

it is possible. When an obstacle is approached, there are two possibilities. First, if an agent can turn in such a way that it can continue moving towards the target, it may do so. Otherwise, it switches into the *obstacle detour* mode: it traverses the obstacle clockwise or counter-clockwise until it reaches a condition when it is possible to continue moving towards the target without hitting an obstacle, or to change the obstacle being detoured. During the detour process, it tracks the cell that is the closest so far to the target. If it is impossible to move towards the target from the closest possible cell, then the target is unreachable, and the agent performs the corresponding action.

4.1 The Model

The first component of the model is a finite description of the part of the field that directly influence the next move of the agent. It consists of:

- information for each of the neighboring cells if it is occupied by an obstacle;
- direction of the agent (north, south, east or west);
- the direction of the target related to the agent;
- information on how the current agent location compares to saved cell in terms of Manhattan distance to the target (closer, farther, at the same distance, cells and directions coincide, only cells coincide).

The first three parts are grouped in a structure called *profile*. The fourth part is stored in the global variable. The part of the model described so far is enough to determine the next move of an agent if the EFSM of the agent is given. All the actions except for "move forward" change the model deterministically. For the "move forward" action, the following components have to be updated non-deterministically: information about obstacles in some of the neighboring cells, the direction to the target and the relation of the saved cell to the current agent location. To reduce the number of false failures, the current profile is saved at the "save position" action, and the last two variants of the latter property can be chosen only if the current and the saved profiles match.

In addition to that, a global bit `detourWall` is used to track whether the agent is detouring an obstacle. We set or clear this bit using heuristic conditions.

The model is implemented in Promela and is verified by Spin [4]. The common part of the model is coded by hand. The part of the model which depends on the actual agent's EFSM is generated by a tool called Stater.[1]

4.2 Weaknesses of the Model

Due to the fact that the model of the agent and the field is finite-state and partially non-deterministic, it can happen that some situations may be produced by the verifier which cannot happen while running the agent on a real field. These situations include:

[1] Available for download at `https://yadi.sk/d/clWWtMrIYhQZJ`.

– mutable field – the visited parts of the field may effectively change;
– infinitely large obstacles;
– infinitely distant target;
– wandering target – the target may change its location;
– wandering saved cell – the location of the saved cell can change in time;
– target in a cell with an obstacle.

These situations cannot happen when evaluating an agent on a real field, so the agent may process them seemingly incorrectly (which does not imply that the agent is incorrect). In out LTL formulae we allow certain forms of incorrect behavior, but ensure it can happen only under impossible conditions.

4.3 LTL Formulae and Theorems

We think there are two possibilities for each agent under verification: it can detour each obstacle either clockwise or counter-clockwise. Technically, it should be possible to construct an agent that can perform both kinds of detours; however, it requires a larger number of states. Accordingly to this idea, we prepared two sets of formulae: the first one is for the clockwise detour, and the second one is for the counter-clockwise one.

The formulae $f0$–$f30$ for the clockwise detour, augmented with their explanation, are available at GitHub.[2] The counter-clockwise versions can be obtained by performing simple "reflective" transformations.

One lemma and four theorems were proven, from which it follows that every EFSM which satisfies the specification (the LTL formulae *f0–f30*) also solves the problem. The theorems and proofs are available at GitHub[3] for the sake of brevity.

4.4 Verification Results

The archive with all necessary Spin models and scripts is available for download[4] for experiment reproduction.

We constructed 1600 models for verification, namely 800 models for each EFSM from [1] using the "clockwise" LTL formula set and 800 modes for the "counter-clockwise" formula set. Verification of all these models took us approximately two days on a 32-core server with AMD Opteron^{TM} 6272 processors.

There were 231 EFSMs which satisfy either clockwise or counter-clockwise LTL formula set. No EFSM satisfied both formula sets, which was expected because any EFSM which satisfies both formula sets traverses every obstacle both clockwise and counter-clockwise. All other 569 EFSMs satisfied none of the formula sets. This does not mean that they are incorrect – they seem to implement a different algorithm (for example, one of them implements BUG-1).

[2] `https://github.com/mbuzdalov/papers/blob/master/2014-hvc-bugs/formulae.ltl`
[3] `https://github.com/mbuzdalov/papers/blob/master/2014-hvc-bugs/proofs.txt`
[4] `https://yadi.sk/d/-orvfVKnYhRFc`

5 Conclusion

We presented an approach that can be used to verify programs in the absence of a finite-state model of the program environment that is precise enough to verify the necessary properties. This approach involves creating a hypothesis about how the verified program works, an intermediate finite-state model and temporal formulae which capture this hypothesis, and finally proving that any program which satisfies the hypothesis performs as expected.

This approach is illustrated on a sample path-planning problem, where constructing a proper counterexample involves creating large unbounded structures. From a previous work [1] we inherited 800 programs in a form of extended finite-state machines which supposedly solve the problem. We were able to prove that 231 of these programs are correct. For other programs proving their correctness should be possible by constructing another hypothesis.

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

References

1. Buzdalov, M., Sokolov, A.: Evolving EFSMs Solving a Path-Planning Problem by Genetic Programming. In: Proceedings of GECCO Companion, pp. 591–594 (2012)
2. Chivilikhin, D., Ulyantsev, V.: MuACOsm: A New Mutation-Based Ant Colony Optimization Algorithm for Learning Finite-State Machines. In: Proceedings of GECCO, pp. 511–518 (2013)
3. Gurov, V., Mazin, M., Narvsky, A., Shalyto, A.: Tools for support of automata-based programming. Programming and Computer Software 33(6), 343–355 (2007)
4. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
5. Kuzmin, E.V., Sokolov, V.A.: Modeling, specification, and verification of automaton programs. Programming and Computer Software 34(1), 27–43 (2008)
6. Lumelsky, V., Stepanov, A.: Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitraty shape. Algorithmica 2, 403–430 (1987)
7. Pingree, P.J., Mikk, E., Holzmann, G.J., Smith, M.H., Dams, D.: Validation of mission critical software design and implementation using model checking (2002), `http://spinroot.com/gerard/pdf/02-1911.pdf`
8. Polikarpova, N., Shalyto, A.: Automata-based Programming, 2nd edn. Piter (2011) (in Russian)
9. Shalyto, A.: Logic control and reactive systems: Algorithmization and programming. Automation and Remote Control 62(1), 1–29 (2001)
10. Tsarev, F., Egorov, K.: Finite State Machine Induction Using Genetic Algorithm Based on Testing and Model Checking. In: Proceedings of GECCO Companion, pp. 759–762 (2011)
11. Yang, C.H., Vyatkin, V., Pang, C.: Model-driven development of control software for distributed automation: a survey and an approach. IEEE Transactions on Systems, Man and Cybernetics 44(3), 292–305 (2014)