Kent Academic Repository

Full text document (pdf)

Citation for published version

Johnson, Colin G. (2007) Genetic Programming with Fitness based on Model Checking. In: Ebner, Marc and O'Neill, M. and Ekart, Aniko and Vanneschi, L. and Esparcia-Alcazar, Anna I., eds. Genetic Programming: 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007, Proceedings. Lecture Notes in Computer Science, 4445. Springer-Verlag, Germany

DOI

Link to record in KAR

https://kar.kent.ac.uk/14594/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

Genetic Programming with Fitness based on Model Checking

Colin G. Johnson

Computing Laboratory, University of Kent
Canterbury, Kent, CT2 7NF, England
C.G.Johnson@kent.ac.uk

Abstract. Model checking is a way of analysing programs and program-like structures to decide whether they satisfy a list of temporal logic statements describing desired behaviour. In this paper we apply this to the fitness checking stage in an evolution strategy for learning finite state machines. We give experimental results consisting of learning the control program for a vending machine.

1 Introduction

In genetic programming (and similar systems for the induction of program code), fitness is typically evaluated by running the programs on a sample of test data. This brings with it a number of difficulties[17], most notably that we cannot formally have confidence in the performance of the system for any input data that has not been used in training. In many cases this does not matter; however, in safety- and mission-critical domains, this is a significant barrier to the adoption of such techniques.

In this paper we give an example of the use of a formal reasoning technique—model checking—as a way of assessing fitness in an inductive automatic programming system. Model checking requires the user to specify the desirable qualities of a system in the form of temporal logic statements about program state; these properties can then be analysed for *all* possible program states. The end result is either a confirmation that the properties always hold, or a counterexample which demonstrates why the system under test does not satisfy the statements.

The paper is structured as follows. Section 2 reviews the literature on the application of program analysis techniques to automatic program induction. This is followed by a section which gives a brief overview of model checking, the analysis technique that we have used in the experiments in this paper. Section 4 discusses how model checking can be used as a fitness measure, and section 5 describes a novel Evolution Strategy for evolving state machines. Section 6 gives the problem specifications that are used in the experiments in the paper, and this is followed by a section which gives results. The paper finishes with some conclusions and ideas for future work.

2 Background - Program Induction with Guaranteed Behaviour

A number of techniques exist which generate computer programs, or other formal executable program-like structures (digital circuits, communications protocols, state machines), by an inductive process which is driven by comparisons between trial solutions and descriptions of the desired behaviour of the system. Perhaps the best known of these is *genetic programming* [2, 20]. This technique applies genetic algorithms to derive executable structures by using a representation which fits well with evolutionary operators such as crossover and mutation. Other evolutionary approaches include *grammatical evolution* [21, 22], where the representation is a simple string of symbols, but this is converted into a complex structure via a grammar; and *evolutionary programming* [10, 11] where evolutionary operators operate directly on a state machine representation. Other heuristic search techniques, such as simulated annealing [7], have been applied to the induction of executable systems. Overall, this area has been termed *search-based software engineering* [6, 13].

Typically such techniques measure the quality of the trial solution by running some instantiations of the system, measuring the results from the system, and comparing those results to the desired behaviour—essentially *testing* the system.

A small number of studies have used measures of fitness that are not dependent solely on testing-like measures of performance, but instead on some form of analysis of the program. In earlier work, these were typically used to guide the search in areas related to performance or evolvability of the solutions: for example measuring length of solutions [26] or using a metric for program complexity [12]. Clearly such measures need to be used alongside a measure of problem-solving quality.

By contrast, a small number of authors have used techniques based on the analysis of the candidate programs to measure ability to solve the problem at hand. Static analysis of programs has been applied to the induction of programs which solve sorting problems [15] and geometric placement problems [17]. Other approaches combine static analysis with data-driven analysis: Keijzer [19] uses static analysis techniques as a preprocessing step, whilst Johnson [18] has applied static analysis to impose safety constraints on programs that are otherwise data-driven.

A general motivation for this kind of approach has been suggested by Partridge and colleagues [23, 24]. He notes that problems that are naturally "data-defined" (e.g. pattern recognition problems) are frequently approached by constructing an artificial "specification bottleneck" which attempts to give a specification to a problem that is best described by giving a number of examples. By contrast, in traditional GP fitness evaluation the opposite problem is sometimes observed: problems for which there is a clear specification are evaluated crudely by giving a number of test cases: this could analogously be termed a "data bottleneck".

The work described in this paper fits into an overall program of work that attempts to measure fitness in terms of the native representation, be it data

or specification. Furthermore, multicriterion optimization methods can be used to solve problems where some aspects of the problem are best described as specifications, and others as data, as illustrated in [18].

3 Background - Model Checking

Model checking [8, 16] is a technique for confirming that a program satisfies a number of conditions. A model checking system takes as input two things. The first of these is a description of some aspect of the system being constructed, expressed as a statement in a temporal logic [9]; that is, statements about how the variables and states in the program change with time. The second input is a description of the system which the user believes should satisfy that description: this might be a computer program, a communications protocol, a state machine, a circuit diagram, et cetera.

The model checking program constructs an abstracted, symbolic representation of the system being analysed, and then uses this representation to decide, in an efficient manner [3], whether the statements always hold in the system, regardless of the control-flow path that is taken through the system. At the end of this analysis, the program reports either a positive result (that the system will always satisfy the statement) or a negative result together with a counterexample which gives a program path under which the statement is not satisfied.

The description language that we use in this study is CTL (Computation Tree Logic) [9]. This consists a number of basic "atomic propositions" (in the examples below, these are labels of states and values of variables in a finite state machine), which can be combined by standard propositional logic connectives and a set of *temporal* connectives which act on propositions (including propositions which themselves contain temporal connectives).

These temporal connectives consist of two components: a description of the *scope* over the future time paths (either A or E) and a description of *when* the proposition that is the argument of the temporal operator holds within that scope (one of G,F,X or U). These have (in informal terms) the following meanings:

A The proposition will hold on **All** paths starting from the current point.
E There **Exists** a path on which the proposition will hold.
G The proposition holds all states (**Globally)** along the path.
F The proposition can be found somewhere (in the **Future**) along the path.
X The **neXt** state satisfies the proposition.
U The proposition holds **Until** a second proposition holds (this is the only binary operator - the rest are unary).

Here a "path" is a sequence of states in time, starting from the current state. Some illustrative examples are given in Figure 1.

This language allows a description of how a process will change with time. Model checking algorithms automatically check whether a particular description of a system satisfies a CTL statement describing the system. The model checking algorithm used below is the SMV system (`http://www.cs.cmu.edu/`

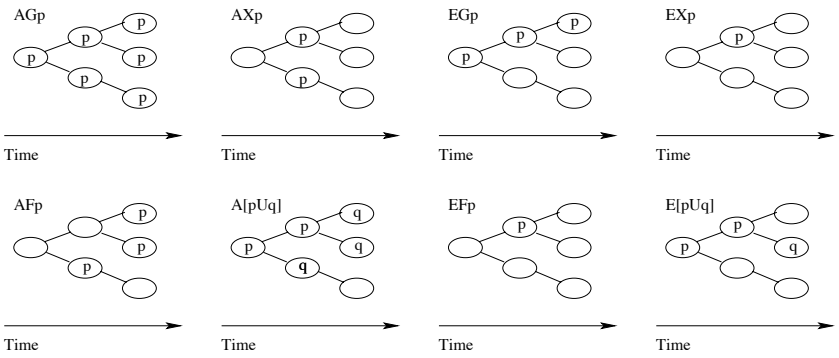

Fig. 1. Illustrative examples of the CTL operators.

`~modelcheck/smv.html`), in particular the version used in the Stuttgart Model-Checking Kit (`http://www.fmi.uni-stuttgart.de/szs/tools/mckit/`).

4 Model Checking as a Fitness Measure

In the experiments below each problem is described as a list of CTL statements that describe the desired properties of the program being evolved. The fitness is measured by the number of statements that are satisfied. To help to smooth out the fitness landscape, some statements are included in the specification lists below that act as *stages* on the way to a complete specification.

5 Methods - Induction of Communicating Finite Automata by an Accumulative Evolution Strategy

In the work in this paper we use *Communicating Finite Automata* (CFA) [4] as the description language for the structures that are evolved. We use a $(1 + \lambda)$ Evolution Strategy with a novel growth-style mutation operator to evolve these structures.

A CFA consists of a number of nodes, which can be labelled from a label-set, linked by a number of directed edges. These edges can contain conditions and actions concerned with global variables and communication channels between automata: e.g. checking whether a variable is set to a particular value or within a range, whether the value of a variable is changed by executing the transition, or whether a variable reads/writes its value from/to a communication channel. Examples are given in Figure 3.

The learning process used in this paper will be referred to as an *Accumulative* Evolution Strategy (AES). This starts with very simple structures, and the most

probably moves that can be made at the mutation stage consist of *adding* items to the structure. Therefore a solution to the problem is built up by accumulating substructures, rather than traditional approaches which begin with structures that are of similar complexity to the final desired structure, and where mutation is typically converting one structure into another of similar size.

The motivation for this variant on traditional evolutionary algorithms is that potential solutions that contain large numbers of arbitrarily connected nodes will fail to satisify any of the fitness conditions. This is because there will as a consequence be a large number of routes through the (nondeterministic) automaton, and therefore there is a large chance that statements such as "X must always be followed by Y" used in the fitness checking will not be satisfied. The aim of our strategy is to build up a solution by conservatively adding new parts to the structure over evolutionary time (a similar point has been made recently by Petrovic [25].

The AES system runs as follows:

INPUT:
  List of labels for nodes
  List of variables and channels
  User model
  List of CTL statements
INITIALISE:
  Create an automaton A consisting of one node labelled "start"
LOOP: until solution found or a fixed number of timesteps completed
  Generate λ mutations $A'_{1..\lambda}$ of A by the following list of processes:
    - with a 0.4 probability, add a new (unconnected) node
      (0.5 probability of label "blank",
      otherwise labelled with a random label from the label set)
    - with a 1.0 probability (always!) add a new link
      (this link has a random label)
    - with a 0.3 probability delete a (randomly chosen) link
    - with a 0.1 probability rename a (randomly chosen) node
    - with a 0.2 probability rename a (randomly chosen) link
  Run the model checker on each statement on each of $A'_{1..\lambda}$
  Count how many statements are satisfied for each of $A'_{1..\lambda}$
  Let the new A be the member of $A'_{1..\lambda}$with the most statements satisfied
ENDLOOP
OUTPUT:
  The best solution found
  The number of generations required to find the best

In the experiments below $\lambda = 20$. The probabilities of performing the various mutations are parameters which have been empirically determined.

No recombination is used in this method at present. This remains an option for future studies—however, recombination has not been heavily used in applications of evolution to finite state automata. One significant reason for this is that there is no clear notion of what a meaningful subroutine is in order to carry our recombination—by contrast, Koza-style tree-based representations [20] have an unambiguous notion of what can be readily swapped between trees (though the

actual role of recombination is controversial). Some attempts have been made [5, 1] to automatically identify functionally coherent modules in automata—such modules could be usefully used as the units of recombination.

wibble

6 Methods - Some Example Specifications

In the experiments below we will generate automata that represent the control systems for coffee vending machines. We use two examples. Each problem consists of two parts: an automaton that represents user behaviour, which is fixed for the given problem; and a specification of the desired machine behaviour, given as a sequence of CTL statements. The specification is then used to measure fitness in the learning process, which learns a machine-automaton which, accompanied by the user-automaton provided, gives a complete description of the system.

6.1 Problem 1

The first problem is a simple machine where the user places a coin into the machine, receives coffee, and the machine then goes into a reset state.

The automaton representing user behaviour in this example is very simple (Figure 2a)—the user can perform one action, putting a coin in the machine, which is represented by adding a value to a *channel* (a list which can be read by other automata), which represents adding a coin into the machine.

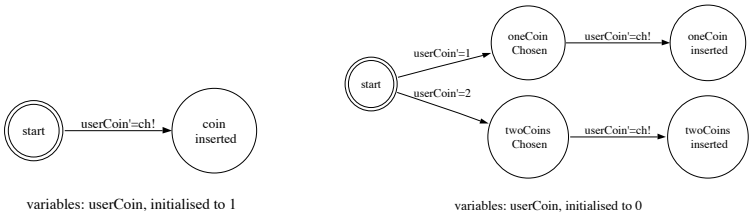

Fig. 2. Automata representing user behaviour: (a) problem 1 (b) problem 2.

The specification of the machine behaviour is given in table 1. Note that some of the behaviours are *staged*, e.g. rules 1–3 are concerned with making certain that certain states are accessible at all, before they are used in specific ways later on. Similarly, rule 5 is the *can* version of the *must* rule in rule 6. The aim of these is to smooth out the fitness space.

Variables	coin, taking values $\{0, 1\}$	
States	coffee	
	reset	
	blank	
Statements	EF("coin=1")	1. We can reach a state where the coin has been inserted
	EF("coffee")	2. We can reach a state labelled "coffee"
	EF("reset")	3. We can reach a state which are labelled "reset"
	AG("userStart"→AF("coffee"))	4. Once the user process has started, coffee must be served
	EF("coffee")&EF("reset")& AG("coffee"→EF("reset"))	5. States coffee and reset can be reached, and whenever coffee has been reached, the machine *can* reset.
	EF("coffee")&EF("reset")& AG("coffee"→AF("reset"))	6. States coffee and reset can be reached, and whenever coffee has been reached, the machine *must* reset.
	EF("coffee")&AG("coffee"→ AX(AG(!"coffee")))	7. Once coffee has been served, we must not serve another coffee
	EF("coin=0")&EF("coffee")& AG("coin=0"→AG(!"coffee"))	8. If no coin has been inserted, we cannot get a coffee

Table 1. The CTL specification for the coffee machine in problem 1.

6.2 Problem 2

The second problem is a machine where the user places either one or two coins into the machine, receives coffee (for one coin) or tea (for two coins), and the machine then goes into a reset state.

The automaton representing user behaviour in this example is given in Figure 2b), and the specification of the machine behaviour is given in table 2.

7 Results

In this section the results of experiments using the Accumulative Evolution Strategy on the two problems are given, and a discussion of the results made.

7.1 Experiment 1

The first experiment consisted of 30 runs of the Accumulative Evolution Strategy on problem 1 with $\lambda = 20$ with the algorithm being stopped after 20 runs if a solution satisfying all the conditions had not been found.

The algorithm found a solution that satisfied all eight conditions in 25/30 runs; in the remaining runs seven conditions were satisfied. In those cases where

Variables	coin, taking values $\{0, 1, 2\}$	
States	coffee	
	tea	
	reset	
	blank	
Statements	EF("coin=1")	1. We can reach a state where one coin has been inserted
	EF("coin=2")	2. We can reach a state where two coins have been inserted
	EF("coffee")	3. We can reach a state labelled "coffee".
	EF("tea")	4. We can reach a state labelled "tea".
	EF("reset")	5. We can reach a state labelled "reset".
	AG("userStart"→AF("coffee"|"tea"))	6. Once the user has started acting tea or coffee must be served
	AG("oneCoinSelected"→AF("coffee"))	7. Once the user has chosen to insert one coin coffee must be served
	AG("twoCoinsSelected"→AF("tea"))	8. Once the user has chosen to insert two coins tea must be served
	AG("oneCoinSelected"→AG(!"tea"))	9. Once the user has chosen to insert one coin tea must not be served
	AG("twoCoinsSelected"→AG(!"coffee"))	10. Once the user has chosen to insert two coins coffee must not be served
	EF("coffee")&EF("reset")& AG("coffee"→EF("reset"))	11. Once coffee has been served we can reset
	EF("coffee")&EF("reset")& AG("coffee"→AF("reset"))	12. Once coffee has been served we must reset
	EF("tea")&EF("reset")& AG("tea"→EF("reset"))	13. Once tea has been served we can reset
	EF("tea")&EF("reset")& AG("tea"→AF("reset"))	14. Once tea has been served we must reset
	EF("coffee")&AG("coffee"→ AX(AG(!("coffee"|"tea"))))	15. Once coffee has been served, no more coffee or tea can be served
	EF("tea")&AG("tea"→ AX(AG(!("coffee"|"tea"))))	16. Once tea has been served, no more coffee or tea can be served
	EF("coin=0")&EF("coffee")& AG("coin=0"→AG(!("coffee"|"tea")))	17. If no coin has been inserted, we cannot get a coffee or tea

Table 2. The CTL specification for the coffee/tea machine in problem 2.

a solution was found, the mean number of generations needed was 6.5 (with standard deviation 4.4).

A number of examples of successful solutions can be found in Figure 3.

7.2 Experiment 2

The first experiment consisted of 30 runs of the Accumulative Evolution Strategy on problem 2 with $\lambda = 20$ with the algorithm being stopped after 30 runs if a solution satisfying all the conditions had not been found.

No run found a solution with all (seventeen) conditions satisfied. The mean number of conditions satisfied was 14 (with a standard deviation of 2). Seven runs found examples where 16 out of the 17 cases were satisfied.

8 Conclusions and Ongoing Work

We have demonstrated how model checking can be used to measure fitness in the evolution of state machines. In the future we intend to apply this to a number of other example problems, including problems which have both specification and data-driven aspects.

An important area for future work in terms of developing the technique will be to develop techniques for smoothing out the fitness landscapes. Ideas in this area include scaling (perhaps dynamically) the fitness contributions of each statement, to encourage the algorithm to search for less well represented statements; using the counterexamples that are returned from failed statements to measure how far the current attempt is from a solution; and estimating the number of paths within the model checking algorithm which do/don't satisfy the statement in order to get away from a simple yes/no response (perhaps using a probabilistic model checking system such as Prism [14]).

References

1. Ricardo Nastas Acras and Silvia Regina Vergilio. Splinter: A generic framework for evolving modular finite state machines. In *Proceedings of SBIA 2004*, pages 356–365. Springer, 2004.
2. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, 1998.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.
4. Thomas L. Casavant and Jon G. Kuhl. A communicating finite automata approach to modeling distributed computation and its application to distributed decision-making. *IEEE Trans. Computers*, 39(5):628–639, 1990.
5. Kumar Chellapilla and David Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation*. IEEE Press, 1999.

6. John Clark, Jose Javier Dolado, Mark Harman, Rob Hierons, Mary Lumkin Bryan Jones, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, and Martin Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.
7. John A. Clark and Jeremy L. Jacob. Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology*, 43(14):891–904, 2001.
8. Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
9. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 955–1072. MIT Press, 1990.
10. D.B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, 1995.
11. Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution*. Academic Press, 1966.
12. Mark Harman and John A Clark. Metrics are fitness functions too. In *Proceedings of the Tenth IEEE International Symposium on Software Metrics*, pages 58–69. IEEE Press, 2004.
13. Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.
14. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: a tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proceedings 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 441–444, 2006. LNCS Volume 3920.
15. Lorenz Huelsbergen. Abstract program evaluation and its application to sorter evolution. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 1407–1414. IEEE Press, 2000.
16. Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.
17. Colin G. Johnson. Deriving genetic programming fitness properties by static analysis. In James Foster, Evelyne Lutton, Conor Ryan, and Andrea Tettamanzi, editors, *Proceedings of the 2002 European Conference on Genetic Programming*. Springer, 2002.
18. Colin G. Johnson. Genetic programming with guaranteed constraints. In Ahmad Lotfi and Jonathan M. Garibaldi, editors, *Applications and Science in Soft Computing*, pages 95–100. Springer, 2004.
19. Maarten Keijzer. Improving symbolic regression with interval arithmetic. In Conor Ryan et al., editor, *Proceedings of the 6th European Conference on Genetic Programming*, pages 70–82, 2003.
20. John R. Koza. *Genetic Programming : On the Programming of Computers by means of Natural Selection*. Series in Complex Adaptive Systems. MIT Press, 1992.
21. Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, August 2001.
22. Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer, 2003.
23. D. Partridge and A. Galton. The specification of 'specification'. *Minds and Machines*, 5(2):243–255, 1995.
24. D. Partridge and W.B. Yates. Data-defined problems and multiversion neural-net systems. *Journal of Intelligent Systems*, 7(1–2):19–32, 1997.

25. Pavel Petrovic. Comparing finite-state automata representation with gp-trees. IDI Technical report 05/2006, Norwegian University of Science and Technology, 2006.
26. Conor Ryan. Pygmies and civil servants. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, pages 243–263. MIT Press, 1994.

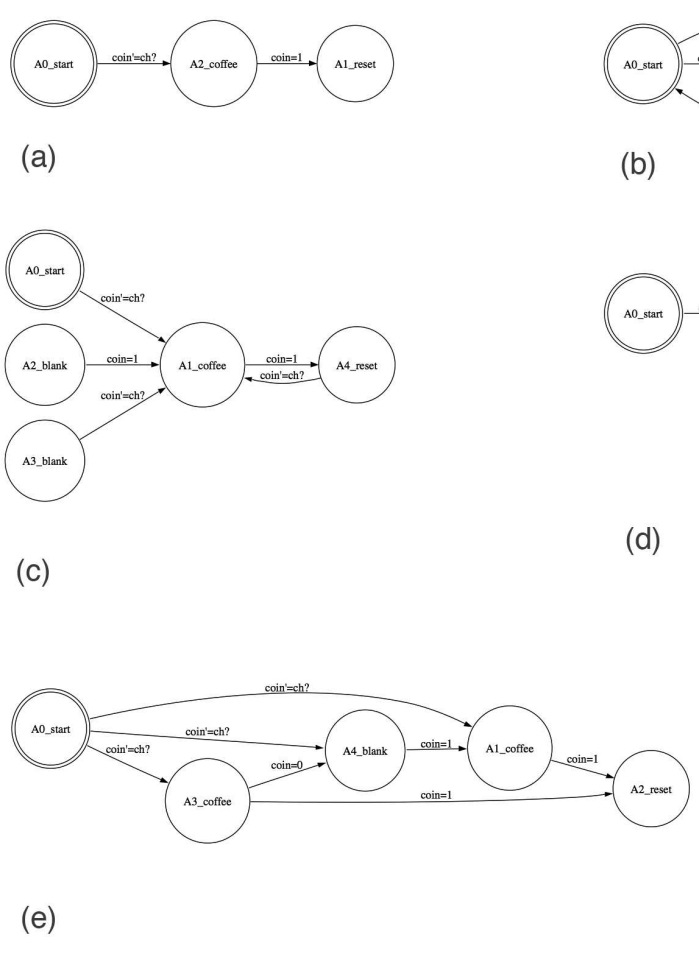

Fig. 3. Example solutions from problem 1