Synthesizing Control for a System with Black Box Environment - Deep Reinforcement Learning Approaches

Itay Cohen (308213883)

Submitted as an assignment report for the RL course, BIU, 2022

Abstract

In this project we aim to improve results of an existing paper about control synthesis for a system with black box environment [1], using two different methods - Deep Recurrent Q-Network (DRQN) and Proximal Policy Optimization (PPO). We compare the results of our approaches with the paper's method. We were able to reproduce or improve the paper's results in each one of the tested experiments.

1 Introduction

Iosti et al. [1] study the deep-learning based synthesis of control for finite state systems that interact with black-box environments. In the studied model, the internal structure of the environment is not provided and its current state is not observable during the execution. In each step, the system makes a choice for the next action, and the environment must follow that choice if the action is enabled. Otherwise, a failed interaction occurs and the system does not move while the environment makes some independent progress. The control enforces the next action of the system based on the available partial information, which involves the sequence of states and actions that occurred so far of the system and the indication of success/failure to interact at each point. The control goal is to minimize the number of times that the system will offer an action that will result in a failed interaction.

The motivation for this problem comes from the challenge to construct distributed schedulers for systems with concurrent processes that will lower the number of failed interactions between the participants. In this case, the environment of each concurrent thread is the collection of all other threads, interacting with it. An alternative approach for constructing schedulers that is based on distributed knowledge was presented in [1].

In their work, Iosti et al. looked at this problem from the reinforcement learning (RL) perspective. RL suggests algorithmic solutions for synthesizing control for systems where the goal is to maximize some accumulated future reward over the interaction with the environment. In the problem described above, the system and the blackbox can be considered as the environment, and the reward will be a numerical representation of the success/failure to interact at each point. In RL, a complete model of the environment does not have to be given, and in this case control can be synthesized through experimenting with it. However, often the current state of the environment is fully observable; this is not the case problem described above. In fact, this problem can be formulated as a Partially Observable Markov Decision Process (POMDP).

More specifically, Iosti et al. used a deep-learning-based variant of the REINFORCE algorithm to control the finite state system. To get the most out of each observation of the environment, they incorporated Recurrent Neural Network (RNN) as a part of their deep learning framework. In their construction, the neural network represents the control of the finite state system. The recurrent neural network's output is the transition that should be offered in the next step. Hence, the neural network directly represents the control's policy. More details on their approach can be found in their paper.

2 Preliminaries

2.1 Formal Description

Our goal is to construct a control for a finite state reactive system that interacts with black-box environment.

System and Environment We study systems that are modeled as finite state automata. Let $A = (S, s_0, T, \delta)$ be the automaton, where

- S is a finite set of states with $s_0 \in G_l$ its initial state.
- T is a finite set of actions (often called the alphabet)
- $\delta : (S \times T) \to S \cup \{\perp\}$ is a partial transition function, where $\perp$ stands for *undefined*. We denote $en(s) = \{t | t \in T \wedge \delta(s,t) \neq \perp\}$, i.e $en(s)$ is the set of actions enabled at the state s. We assume that for each $s \in S$, $en(s) \neq \phi$.
- An optimal component is a probabilistic distribution on the selection of actions $d : S \times T \to [0,1]$ where $\sum_{s \in S} d(s,t) = 1$. Then, the model is called a markov chain. In our case, the distribution over the transitions is *uniform* for each $s \in S$.

The asymmetric combination of a system and an environment $A^s \lceil A^e$ involves the system automaton $A^s = (S^s, s_0^s, T^s, \delta^s)$ and the environment automaton $A^e = (S^e, s_0^e, T^e, \delta^e)$ where $T_s \cap T_e \neq \phi$. The two components progress synchronously starting with their initial state. The system offers an action that is enabled from its current state. If this action is enabled also from the current state of the environment automaton, then the system and the environment change their respective states, making a successful interaction. If the action is not currently enabled by the environment, the interaction fails; in this case, the system remains at its current state, and the environment chooses randomly some enabled action and moves accordingly. After a failed interaction, the system can offer the same or a different action.
Formally,

$$A^s \lceil A^e = (S^s \times S^e, (s_0^s, s_0^e), T^s \times T^e, \delta)$$

where

$$\delta((s^s, s^e), (t^s, t^e)) = \begin{cases} (\delta^s(s^s, t^s), \delta^e(s^e, t^s)) & \text{if } t^s \in en(s^e) \\ (s^s, \delta^e(s^e, t^e)) & \text{otherwise} \end{cases}$$

An environment with a probabilistic distribution on selecting actions makes a probabilistic choice only if the action offered by the system is currently disabled. In our case, where the system's action is disabled by the environment, the distribution $d(s^e, \cdot)$ is *uniform*.

2.2 Example

Consider the system in Figure 1 (left) and its environment (right). This system can always make a choice between the actions a, b and c, and the environment has to agree with that choice if it is enabled from its current state. Remember that the system is unaware to the environment's internal state. If the system selects actions according to $(bca)*$, then the environment can follow that selection with no failures. On the other hand, if the system selects actions according to $(baa)*$, the system will never progress, while the environment keeps changing states. Our goal is to construct a control that restricting the system's actions at each step such that the number of failed interactions will be minimal.

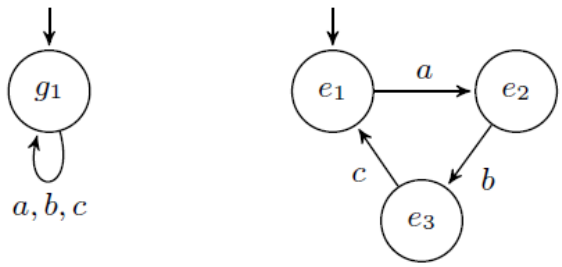

Figure 1: 'permitted' scenario

3 Experiments

The paper provides six different experiments - the first four are relatively simpler than the latter two. We tested our models against the four simpler experiments and the last complex experiment. The last complex experiment (*cycle scc*) assumed to be the most difficult one, since Iosti et al. had the highest average failure rate in this one. The following table exhibits the properties of the tested experiments. The formal description of the experiments' automatons can be found in the appendix.

Experiment	**# Environment States**	**Failure Rate (%) of Optimal Policy**
Permitted	3	0
Schedule	4	0
Cases	5	1.5
Choice-scc	25	1.5
Cycle-scc	25	1.5

Table 1: Experiments list. Failure percentages are per episodes with 200 timesteps.

4 DQN based Approach

As a first method, a DQN based approach was tried. Since our problem is defined as POMDP, the state space is only the possible observations of the system component. Moreover, we would like to use previous system states and actions to make intelligent decisions in the future. Therefore, we chose to add a recurrence component to the deep learning architecture (RNN) which captures the history of the execution. The main challenge was to add the RNN (LSTM in our case) to the DQN architecture, and find an optimal way to train it. The recurrent variant of DQN is also known as DRQN [2].

4.1 State Representation

Since there are no graphics in the presented problem, CNN is redundant. Instead, we take a feature engineering based approach and compose the state vector out of the available information at each time step. Another component of the state vector is the succinct representation of the sequence history, given by the LSTM. In our implementation, the LSTM layer replaces the CNN layer. Then, an MLP layer is used. More specifically, the DRQN state consists of two parts:

Short Term History consists of the last system's state (the automaton state), the last system action offered to the environment component, and whether or not the action was successful. These three parts are encoded as a single vector.
Long Term History a finite representation of the past actions and states of the system component, given by the LSTM.

4.2 Reward Modeling

The reward function of this DQN approach is very intuitive. The reward value for a successful interaction is 1, and -1 for a failed interaction.

The optimal policy in each one of the experiments induces a behaviour of consecutive successes. Therefore, we tried to prioritize consecutive successes through the rewards. Different variations of this approach were tested:

- **Multiplication Factor** for each reward streak with $length \geq 4$, the reward of success grows in a geometric fashion.
- **Bigger Constant Reward** for each reward streak with $length \geq 4$, the reward of success changes to a bigger constant value.
- **Linear Growth** for each reward streak with $length \geq 4$, $reward = 1 + \frac{streak\_length}{5}$

None of the variations mentioned above yielded better results than the naive approach.

4.3 The Training Process

DQN is an offline RL algorithm, as a result it uses an experience replay of its previous samples. As mentioned in [2], in DRQN most of the training process remains the same, except of the way items are sampled from the replay memory. With a recurrent component, the addressed problem is no longer an MDP. The next chosen action depends on the whole sequence of states, actions and rewards. Therefore, we can no longer randomly sample single transitions from the replay memory. We need to change the optimization phase of DQN to train the LSTM properly. We tested two different approaches:

Sequential Updates: full episodes are selected randomly from the replay memory and updates begin at the beginning of the episode and proceed forward through time to the conclusion of the episode. The targets at each time step are generated from the target Q-network. The LSTM's hidden state is carried forward throughout the episode. The code also supports an optimization of a batch of sequences at once.
Sequence-Fractions Updates: Batch of fractions of sequences are sampled from the replay memory in the optimization phase. In this approach, the LSTM hidden states are also saved in the replay memory since a fraction may start from the middle of a sequence.

Due to its superior performance, the first approach was chosen.

4.4 Successful Modifications

We added a few more modifications to improve the results. First and foremost, we added a simplified prioritized replay memory for DRQN. The optimization step includes sampling of full episodes, and rewards of each one of them is known. Therefore, reward-rich episodes can be prioritized over those with lower rewards. The different episodes were categorized into deciles, and different probabilities are assigned to each decile. We tried a few sampling distributions, the chosen deciles sampling distribution can be found in the appendix.

It is a well known fact that exploration is critical to the performance of the DQN algorithm. At the beginning, we tried the epsilon greedy approach. After several attempts, we found this approach relatively sensitive to small changes. In contrast, **Boltzmann Exploration** was found to be more robust. With this approach, the optimal policy could be reached more frequently. We let the *temperature* parameter of this method gradually decrease until reaching a certain threshold, to enable exploration with small probability in the late training steps.

The results were also influenced by another hyperparameter - *target network update frequency*. At first, the DQN approach was tested with a relatively high frequency of updates. The target network was updated every 10 episodes. As time went on, it appeared that a lower frequency of 50 episodes contributed to the control getting closer to the optimal policy with greater probability.

The last modification that caused a dramatic impact to the results was *early stopping*. We started evaluating the policy every 10 episodes. It was decided that if the policy passed a certain threshold of evaluations three times in a row, the training would end.

4.5 Ineffective Modifications

The following approaches were tested but did not produce any positive results:

- Optimization phase after a whole episode rather than a single step.
- Sampling a fractions of episodes including their hidden states.
- In a few experiments the network seemed somewhat unstable. To mitigate this, we implemented the *Double-DQN* method [3], and *Entropy Regularization* [4] however the results have only gotten worse. The implementation of these approaches can be found in the link provided in the appendix.
- Increasing the training sequence length, and the amount of training episodes.
- Concatenation of the last two short term history vectors, to emphasize the importance of the latest data arrives.

- Modifying of the network's architecture - added two linear layers before the LSTM layer.
- Increasing the batch size of sampled episodes in the optimization phase.
- Changing the loss function from Huber-loss to MSE.
- Changing the optimizer type from RMSProp to Adam.

4.6 Results

For each experiment, we repeated the process 10 times and averaged the minimal failure percentage of a sequence. The process consists of a training and evaluation phase. The training phase had 3000 episodes, each one has a length of 50. The evaluation phase had 100 episodes, each one has a length of 200. The evaluation part of the experiments is identical to the paper.

Experiment	Paper's Best Result	DRQN Results
Permitted	0	0
Schedule	0	0
Cases	1.5	1.5
Choice-scc	4.5	1.5
Cycle-scc	33.5	8

Table 2: Failure rates (%) - paper vs. DRQN

5 Proximal Policy Optimization Based Approach

Even though the DRQN results were promising, the fine-tuning of this algorithm was tedious. It took a few weeks to add the correct modifications and to find the optimal hyperparameter values. Therefore, we searched for an actor-critic based algorithm which requires less hyperparameters tuning. Proximal Policy Optimization was chosen for this purpose.

5.1 Algorithm Overview

Proximal Policy Optimization (PPO) is an on-policy actor-critic based algorithm, developed by OpenAI [5]. This method considered to be stable and reliable comparing to other deep RL methods, and can be helpful in a wide variety of settings. This algorithm has two main differences when compared to others actor-critic algorithms. The first difference has to do with its objective. It uses the *Clipped Surrogate Objective* - which replaces the traditional objective of vanilla policy gradient methods. The second one has to do with the way the policy is updated. Unlike vanilla policy gradient methods, and because of the clipped surrogate objective function, PPO allows you to run multiple epochs of gradient ascent on the given samples without causing destructively large policy updates. This allows to squeeze more out of the data and reduce sample inefficiency. Moreover, this algorithm has a relatively small amount of hyperparameters which implies that less fine-tuning is required.

5.2 Implementation Details

It was mentioned before that we would like to use previous system states and actions to make intelligent decisions in the future. In this algorithm, incorporating an RNN was a quite challenging task. Therefore, we tested the original architecture of PPO, without the CNN layers which are redundant in our scenario. We found out that among all the experiments, only the *'cases'* scenario significantly relies on the long term history. The four other scenarios are able to rely solely on the short term history to learn an optimal policy.

The algorithm's skeleton was forked from [6]. The main changes of the skeleton were in terms of changing the state space and the action space to discrete (instead of continuous). Except of the CNN layer removal, the MLP layers of the actor and critic networks stayed the same. The reward function is the same as for the DQN based approach. The optimization algorithm was decided to be Adam.

5.3 Hyperparameters

- Episode Timesteps - the maximal length of an episode within a single rollout.
- Batch Timesteps - the maximal timesteps allowed in a single rollout. Timesteps are four-tuples of (state, action, reward, next-state)
- Discount Factor - denoted by γ.
- Iteration Updates - number of allowed epochs of gradient ascent on the given samples in the batch (rollout).
- Learning Rate - the learning rate of the optimization algorithm used (Adam proved to be better in this case).
- Clipping Factor - the epsilon clipping factor of the Clipped Surrogate Objective.
- Total Timesteps - the total number of four tuple that would be required for the training phase. The training process stop when number is exceeded.

5.4 Results

For each experiment, we repeated the process 10 times and averaged the minimal failure percentage of a sequence. The process consisted of a training and evaluation phase. The training phase had 50000 episodes, each one has a length of 100.

Experiment	Paper's Best Result	PPO Results
Permitted	0	0
Schedule	0	0
Cases	1.5	1.5
Choice-scc	4.5	1.5
Cycle-scc	33.5	3

Table 3: Failure rates (%) - paper vs. PPO

The graph below shows the average episodic rewards as a function of number of training episodes for *'cycle scc'* scenario. Each episode has a length of 100, and the mininum failure rate is 3%. Therefore, the maximum average accumulated rewards is up to 94 out of 100.

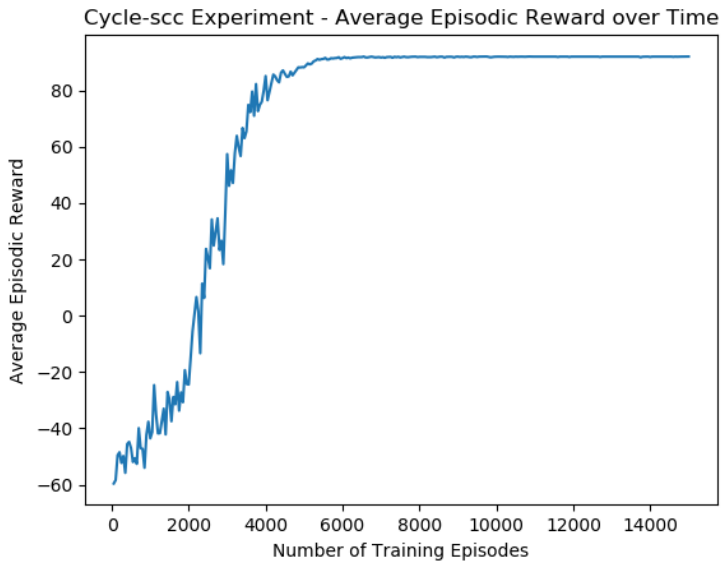

Figure 2: 'cycle-scc' scenario - rewards as a function of training episodes

6 Conclusions

The two tested approaches are deep reinforcement learning based, like the method described in the paper. They proved to be just as efficient and even better in learning the scenarios described above, compared to the paper. In fact, there is a significant improvement in the results of the complex scenario. In this scenario PPO almost reaches the exact optimal policy on average.

It is interesting to see that in both algorithms we found a uniform set of hyperparamters that performs well across all experiments. On the other hand, Iosti et al. found different sets of hyperparameters for different scenarios.

PPO proved to be much more robust and simpler to tune. It has less hyperparameters, but it still performed well and achieved the best results in the complex scenario. PPO also taught us that all the experiments but one did not require the use of long term history. A future work in this direction may be implementing a recurrent variant of PPO and test it on new scenarios that require long term history to learn an optimal policy.

References

[1] S. Iosti, D. Peled, K. Aharon, S. Bensalem, and Y. Goldberg, "Synthesizing control for a system with black box environment, based on deep learning," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2020, pp. 457–472.

[2] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *2015 aaai fall symposium series*, 2015.

[3] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[4] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, "Bridging the gap between value and policy based reinforcement learning," *Advances in neural information processing systems*, vol. 30, 2017.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[6] (2020) Ppo for beginners. [Online]. Available: https://github.com/ericyangyu/PPO-for-Beginners

[7] (2022) Deep control learning code. [Online]. Available: https://github.com/itay99988/Deep-Control-Learning

7 Appendix

7.1 Code

The code for this project can be found on GitHub [7].

7.2 DRQN Hyperparameters

Hyperparameter	Value
Batch Size	2
Discount Factor	0.88
Target Network Update Frequency	50
RNN Hidden Dimension	10
MLP Hidden Dimension	7
Learning Rate	0.01
Initial Temperature	8000
Temperature Thershold	5
Training Episodes	3000
Training Episode Length	50
Replay Memory Deciles Weights	(1, 1, 1, 2, 2, 2, 2, 4, 4, 4)

Table 4: DRQN Hyperparameter List

7.3 PPO Hyperparameters

Hyperparameter	Value
Batch Timesteps	5000
Training Episode Length	100
Discount Factor	0.99
Iteration Updates	10
Learning Rate	0.01
Clipping Factor	0.2
Total Timesteps	5000000
MLP Hidden Layer Dimension	10

Table 5: PPO Hyperparameter List

7.4 Experiments Figures

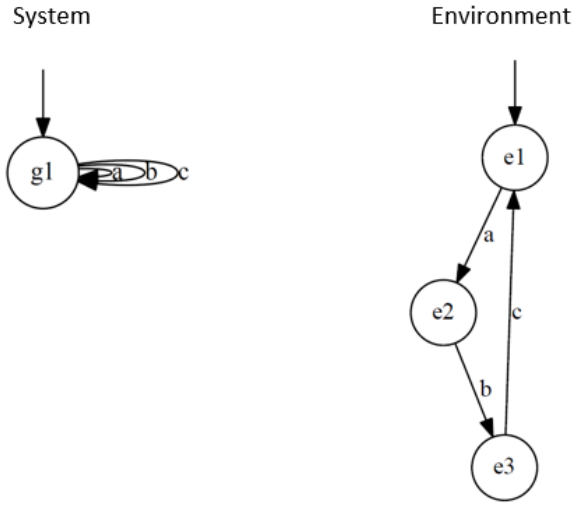

Figure 3: 'permitted' scenario

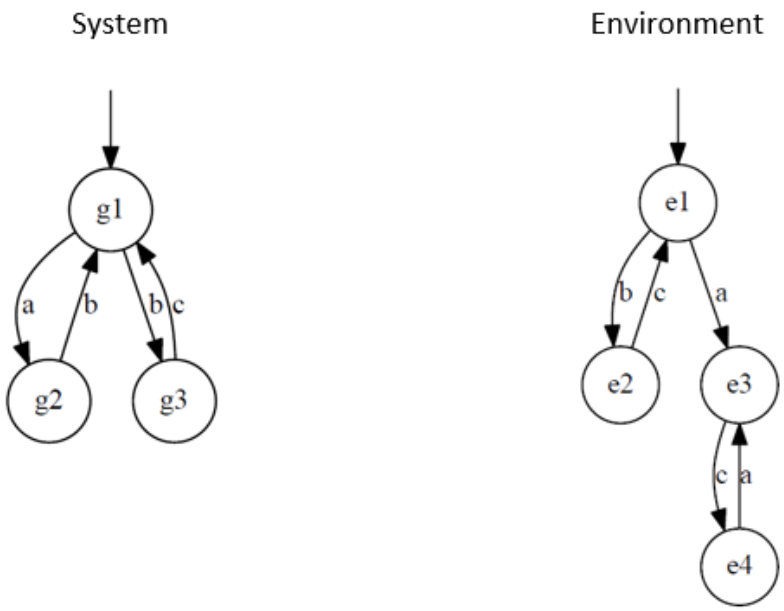

Figure 4: 'schedule' scenario

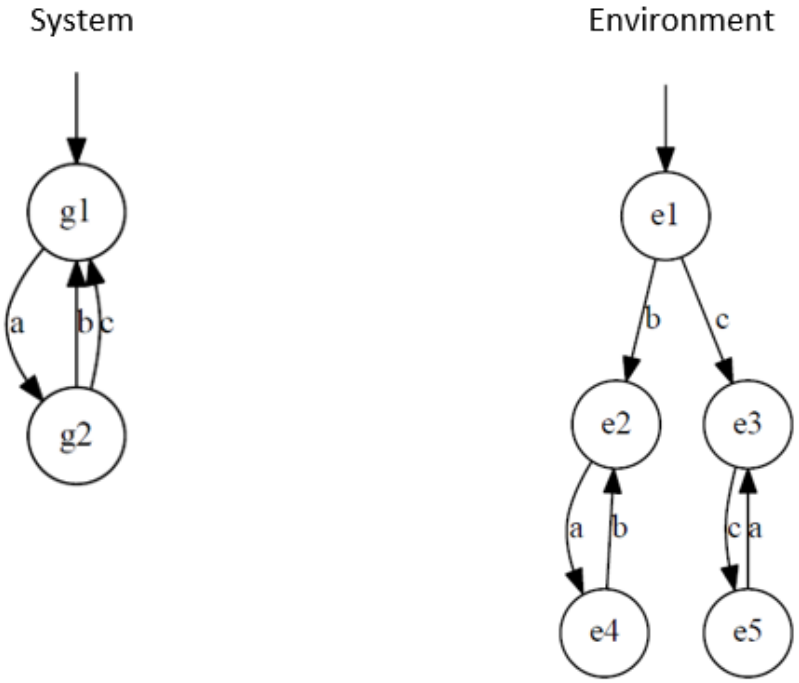

Figure 5: 'cases' scenario

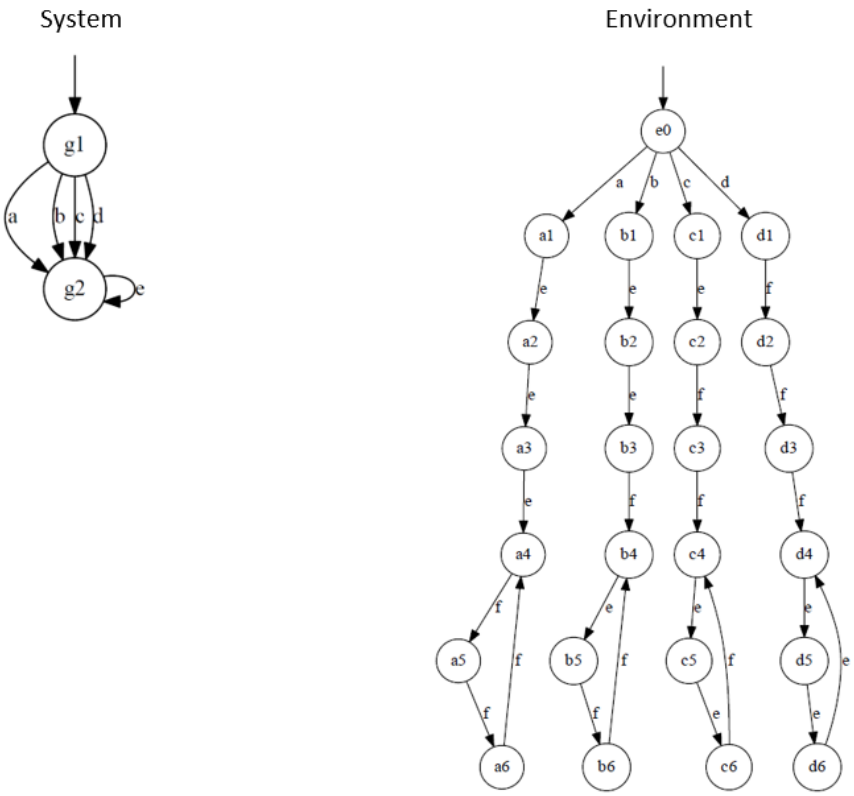

Figure 6: 'choice-scc' scenario

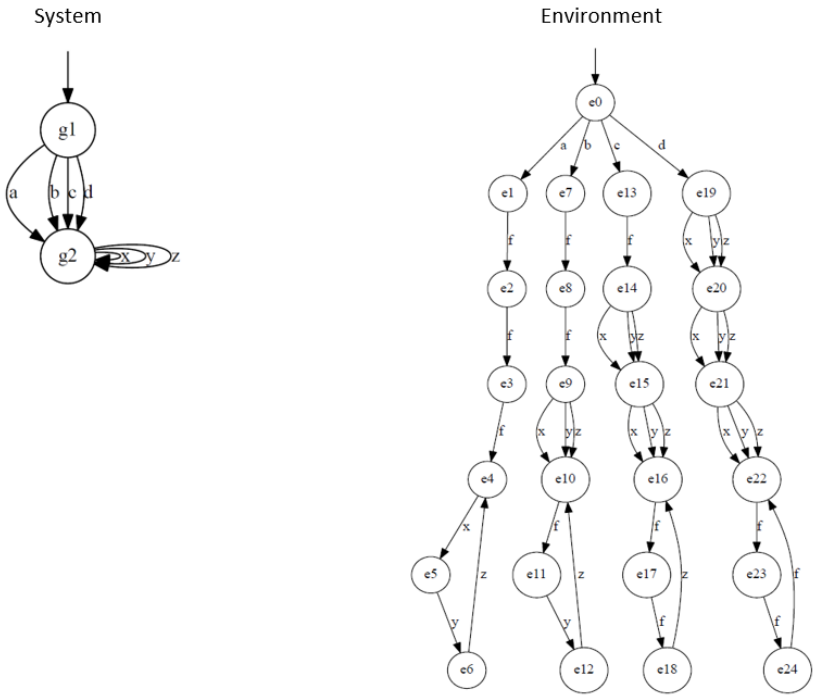

Figure 7: 'cycle-scc' scenario