Control Improvisation for Cyber-Physical Systems

CNS-1646208

Daniel J. Fremont, Xiangyu Yue, Tommaso Dreossi, Shromona Ghosh,
Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia

University of California, Berkeley

Motivation

Introducing randomness into the behavior of a system can enhance variety, robustness, or unpredictability. Some examples:

- **Protocol fuzz testing:** We want to generate many different packet sequences, while conforming to the protocol (perhaps only most of the time).
- **Exploration:** A robot moving in an unknown environment can use randomness to increase coverage of the space or reduce exploration bias.
- **Robotic surveillance:** Using a random patrol route makes the robot's future location harder to predict.

However, adding randomness should not compromise safety and correctness. Control Improvisation (CI) is a framework for synthesizing randomized systems with formal guarantees. Here, we study two applications of CI to cyber-physical systems:

- **Randomized robotic planning** in an adversarial environment;
- **Generating synthetic data** to test or train an autonomous car.

Reactive Control Improvisation

To enable randomized planning in an adversarial environment, we defined a reactive version of control improvisation.

In RCI, the system σ and environment (adversary) τ alternate picking symbols from a finite alphabet Σ, building up a word of length n. Let $P_{\sigma,\tau}(w)$ be the probability of obtaining the word w.

An *improvisation* is any word $w \in \Sigma^n$ satisfying a hard constraint $\mathcal{H}$, and I is the set of all such words. An improvisation is *admissible* if it also satisfies a soft constraint $\mathcal{S}$, and A is the set of all admissible improvisations.

Given an *RCI instance* $\mathcal{C} = (\mathcal{H}, \mathcal{S}, n, \epsilon, \rho)$ with $\epsilon \in [0,1]$ and $\rho \in (0,1]$, a strategy σ is an *improvising strategy* if for every adversary τ:

- $P_{\sigma,\tau}(I) = 1$ (hard constraint always satisfied)
- $P_{\sigma,\tau}(A) \geq 1 - \epsilon$ (soft constraint usually satisfied)
- $\forall w \in I,\ P_{\sigma,\tau}(w) \leq \rho$ (sufficient randomness)

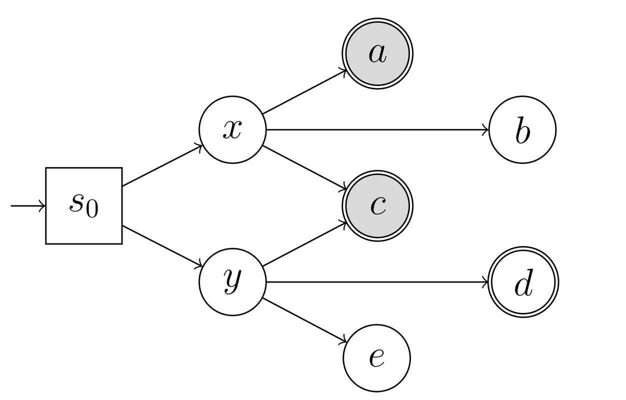
If there is an improvising strategy, $\mathcal{C}$ is *feasible*; an *improviser* for $\mathcal{C}$ is a probabilistic algorithm implementing an improvising strategy.

Reachability Game Example

square = adversary-controlled state
doubled = goal for hard constraint
shaded = goal for soft constraint

With $\epsilon = \rho = 1/2$, this is feasible.
An improviser:

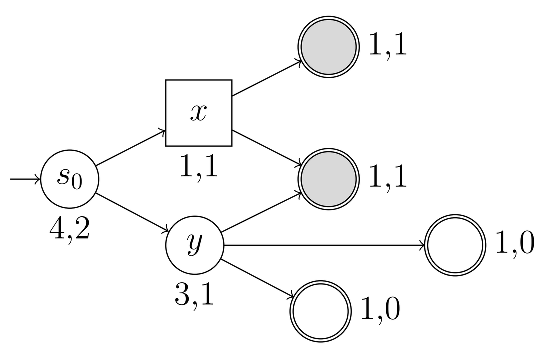
- $x \to a, c$ with equal probability
- $y \to c, d$ with equal probability



Whatever the adversary does, we always reach a doubled state, we reach a shaded state with at least probability $1/2$, and no single path has more than probability $1/2$.

Improviser Construction

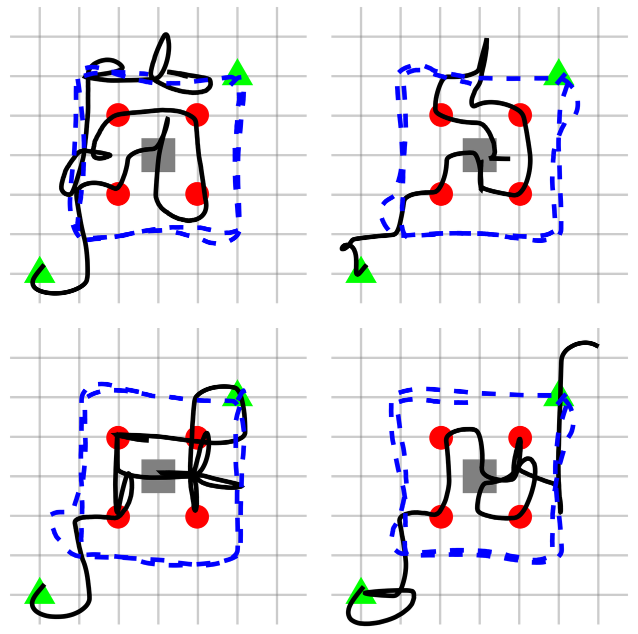
We show how to construct improvisers by doing a random walk weighted by the number of ways to satisfy the hard and soft constraints.
For example, moving to x and y with probabilities $1/4$ and $3/4$ we can achieve $\rho = 1/4$ and $\epsilon = 2/3$.



For reachability and safety games, our construction can be implemented efficiently.

Randomized Robotic Planning

We used our RCI algorithm to synthesize a planner for a surveillance drone (black) that visits the 4 circled locations while avoiding collisions with another, potentially adversarial drone (blue). We imposed a soft constraint saying that 3/4 of the time, the drone should not redundantly visit one of the circles. At right, 4 runs against the same adversary illustrate the randomness of the controller.



Generating Synthetic Data

- Collecting, preparing, and labeling real-world data is slow and expensive; furthermore, it can be hard to observe corner cases that are rare but necessary to test against (e.g. a car accident).
- Synthetically generated data can be produced in bulk with correct labels.
- However, generating meaningful data is difficult since the input space of ML systems is often large and unstructured. Images of 12 cars placed randomly on a road, facing random directions are not very useful.
- We want scenes that are interesting for testing or training purposes.
- Inspired by CI, we propose to guide data generation with hard and soft constraints encoded in a domain-specific programming language, Scenic.

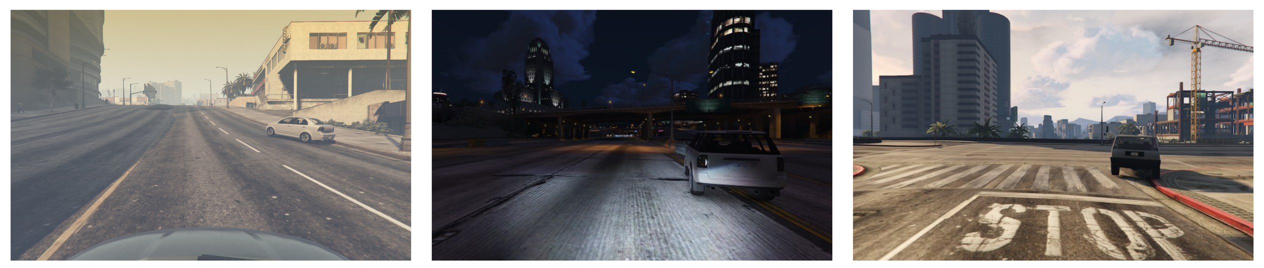
The Scenic Scenario Description Language

Scenic is a probabilistic programming language defining distributions over *scenes*, which are configurations of physical objects. For example, here is a Scenic scenario describing a badly-parked car, with 3 scenes generated from it:

```
from gta import Car, curb, roadDirection

ego = Car

spot = OrientedPoint on visible curb
badAngle = Uniform(1.0, -1.0) * (10, 20) deg
Car left of (spot offset by -0.5 @ 0),
    facing badAngle relative to roadDirection
```



With Scenic, far more complex scenarios can be described easily. Two examples:

5-car platoon (10 lines of Scenic):

Bumper-to-bumper traffic (25 lines):

The configurations generated by Scenic can be fed into a simulator to produce images, LiDAR data, etc. For our experiments we used Grand Theft Auto V to render images for a car detector neural network.

Training on Hard Cases

A difficult case for car detection is when two cars overlap in the image. We can generate such scenes using Scenic:

```
from gta import Car

ego = Car with roadDeviation (-10, 10) deg

c = Car visible,
        with roadDeviation (-10, 10) deg

leftRight = Uniform(1.0, -1.0) * (1.25, 2.75)
Car beyond c by leftRight @ (4, 10),
    with roadDeviation (-10, 10) deg
```

We can significantly improve the performance obtained using a state-of-the-art dataset [1] (synthesized with the same simulator) by mixing in these difficult images, keeping the total size of the training set fixed:

Training Data (5k images)	Generic Test Set Precision	Generic Test Set Recall	Overlapping Test Set Precision	Overlapping Test Set Recall
100% generic	72.9	37.1	62.8	65.7
95% generic, 5% overlapping	73.1	37.0	**68.9**	**67.3**

Including the overlapping images dramatically improves performance on such images, without hurting (and in fact improving!) performance on generic images.

[1] Johnson-Roberson et al., *Driving in the Matrix*, ICRA 2017.

Generalizing a Known Failure

We can use Scenic to reproduce a known failure case, then generalize it for testing or retraining. Here, the neural network misclassifies one car as three:

```
from gta import Car, EgoCar, CarModel, CarColor

param time = 12 * 60        # noon
param weather = 'EXTRASUNNY'

ego = EgoCar at -628.8 @ -540.6,
             facing -359.2 deg

Car at -625.4 @ -530.8,
    facing 8.3 deg,
    with model CarModel.models['DOMINATOR'],
    with color CarColor.byteToReal([187, 162, 157])
```

We can generalize this scenario in different directions to discover what features contributed to the misclassification. Here are images generated from 3 different generalizations:

Add noise: | Change car model: | Change global position:

We can then write a more general scenario that captures the cause of the failure, retraining the network without overfitting to the original image.

Exploring System Performance

Scenic can be used to write specialized test sets to evaluate system performance under different conditions. For example, four cars in good or bad lighting and weather:

Future Work

Theory of Control Improvisation:

- RCI over unbounded or infinite words, for robotic planning;
- More complex randomness constraints — e.g. bounds on entropy — directly controlling diversity or unpredictability;
- Control improvisation over continuous signals, for test generation.

Extensions of Scenic:

- encoding dynamics to generate videos instead of static scenes;
- describing 3D scenes;
- allowing users to extend the language by defining custom specifiers;
- interfacing Scenic to other simulators, e.g. CARLA.

Conclusions

- Reactive Control Improvisation is a framework for synthesizing systems with random but controlled behavior.
- We showed RCI problems can be efficiently solved in many practical cases, and used it to synthesize a planner for a surveillance drone.
- Scenic is a probabilistic programming language for specifying distributions over configurations of physical objects.
- Scenic can generate synthetic data useful for analyzing ML-based perception systems:
  - creating specialized test sets to explore system performance;
  - improving training effectiveness by emphasizing difficult cases;
  - generalizing from individual failure cases to discover the cause of the failure and to construct broader scenarios suitable for retraining.

Bibliography and Acknowledgements

Fremont and Seshia, *Reactive Control Improvisation.* CAV 2018.

Fremont et al., *Scenic: A Language for Scenario Specification and Scene Generation.* PLDI 2019.

This work is supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1106400, NSF grants CNS-1646208, CCF-1139138, and CNS-1739816, DARPA under agreement number FA8750-16-C0043, the DARPA Assured Autonomy program, and TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.