Exploiting While Exploring: Effective Bug Discovery in Unit-Level Verification

Hongsup Shin

Arm, Ltd.

Author bio

- Hongsup Shin
- Data scientist / ML Researcher at Arm (3 years)
- Developing ML applications for verification
- PhD in neuroscience

Simulation-based hardware verification

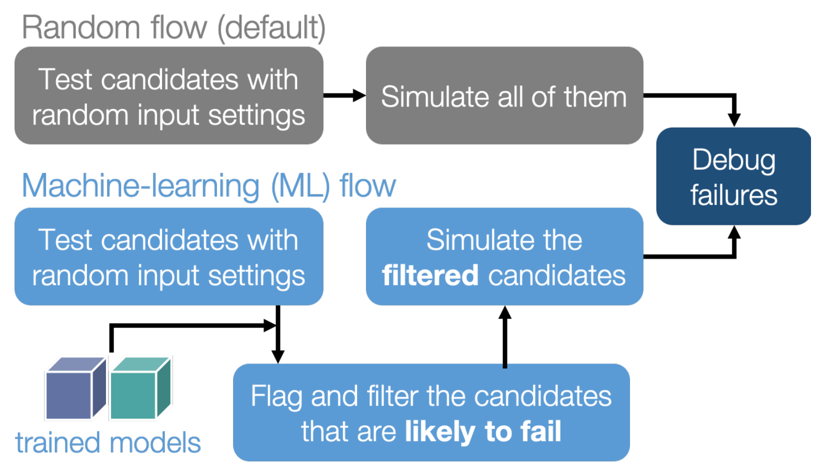
- 60-70% of the cycle dedicated to verification
- Exhaustiveness doesn’t scale well with design complexity
- Random-constraint simulation: possible to direct tests with constraints but still non-deterministic
- Simulation stimuli examples
  - Binary on/off switch of a setting
  - Probabilistic definition of numerical constraints (ranges)
- Previously-simulated tests are not well utilized by verification engineers.

Failures in unit-level verification and ML

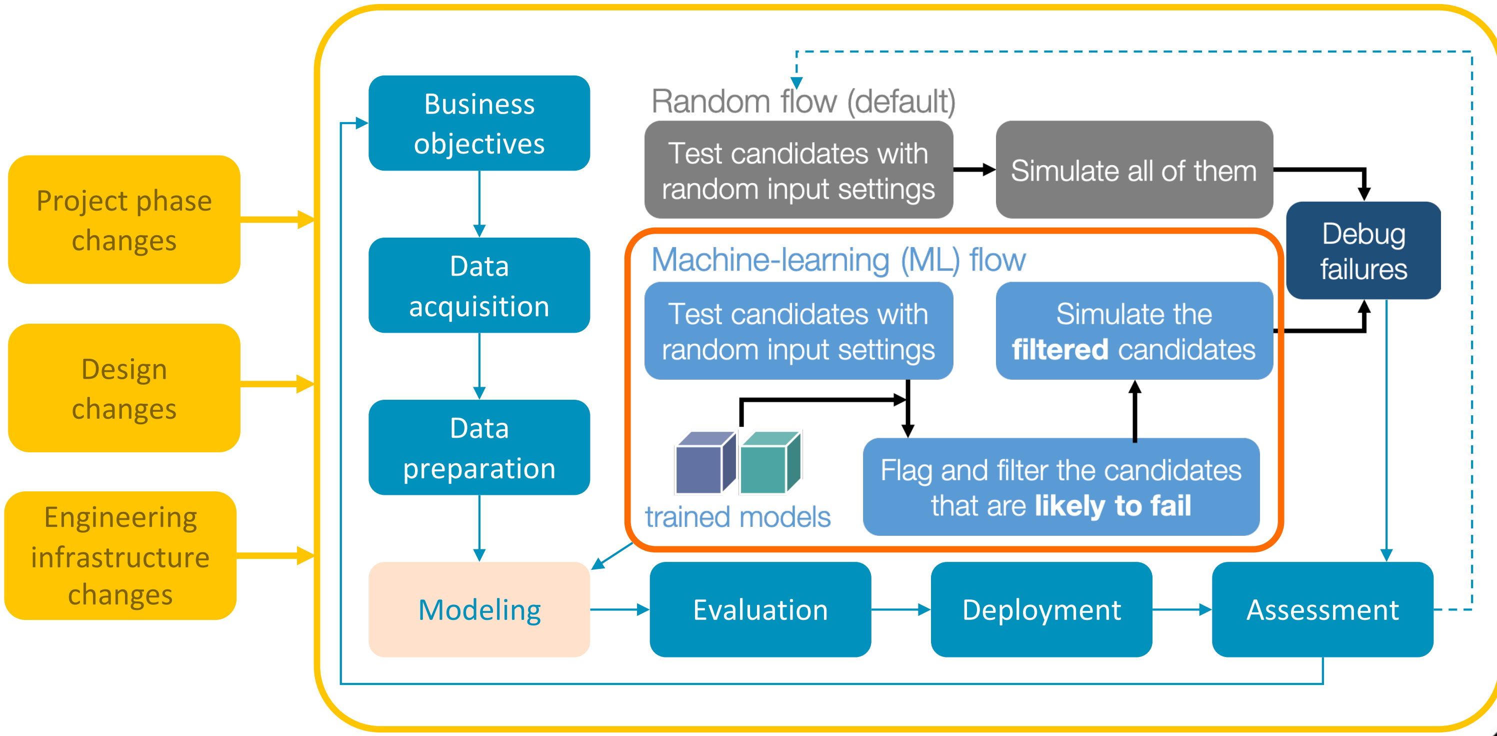
- Can we use previously run tests to predict which tests will fail in advance?
- Failures (=bugs): with a set of given input settings, HDL-level simulations produce undesirable outputs
- ML approaches in literature suggest exploratory algorithms (e.g., reinforcement learning, evolutionary algorithms)
- Lack of concrete examples especially that address the details of ML engineering in deployment
- General ML engineering challenges
  - Stochastic nature of test bench
  - Frequent changes to the design and test bench
  - Class imbalance
  - Fast inference time

Our approach: tagging bug-prone tests w/ ML

- Train the ML models based on previously simulated tests
- Use the existing testbench infrastructure to generate a large set test candidates (e.g., input settings)
- The trained models make a binary prediction for each candidate (pass or fail)
- Only the candidates that are flagged as failure are simulated.

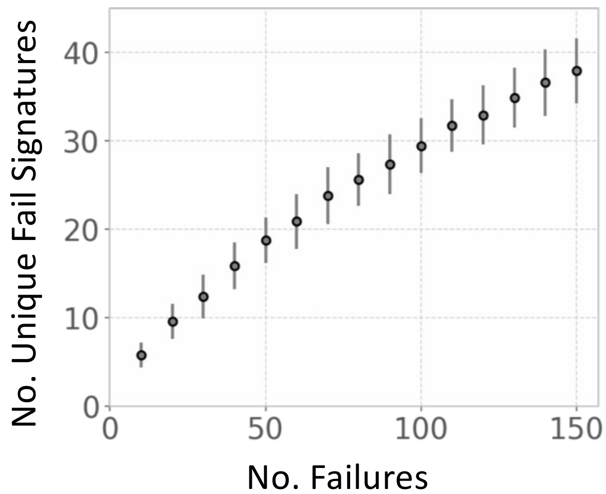


Overview of the bug-hunting application



Training data

- 100k simulated tests (2-week worth)
- From a specific unit of a microprocessor with a specific test scenario
- Several hundreds of input settings (features)
- Target: <u>binary pass/fail</u> and fail signatures (hash function output from failure logs)
- Extensive and semi-automated data preprocessing

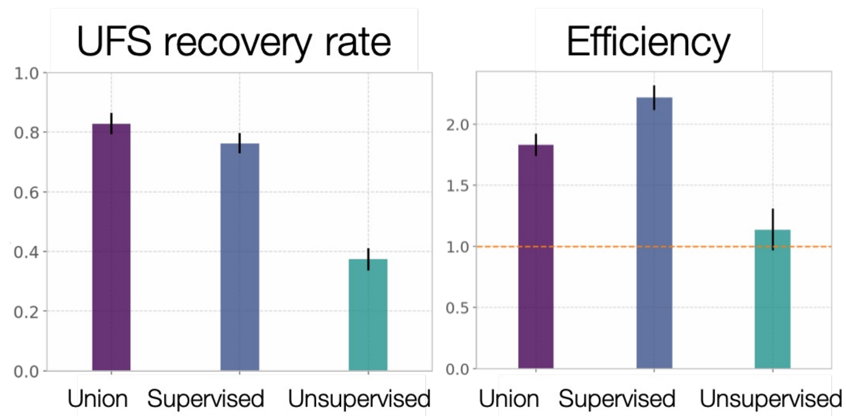


Metrics

- Unique failure signature (UFS) recovery rate
  - Similar to recall
  - Ratio between the no. of UFS found and the no. of total UFS in the validation set
  - Higher the better
  - 1 = the models recovered all UFS
- Efficiency
  - How efficient the ML flow is compared to the baseline (the random flow)
  - E.g., efficiency of 2 = the models can capture x2 failures compared to the baseline when the same number of tests are run
  - Higher the better

Models and deployment

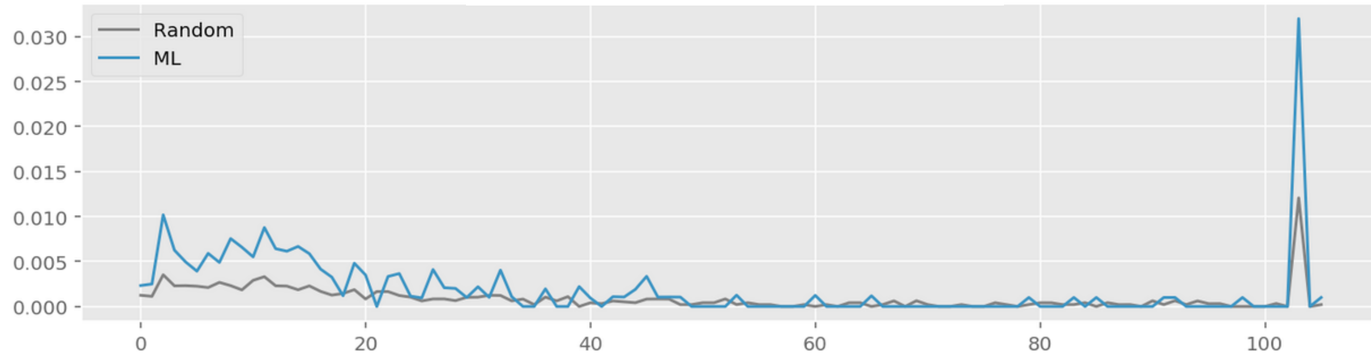
- An ensemble of supervised and unsupervised models
  - Supervised: identify failures similar to the previous ones (gradient boosting)
  - Unsupervised: find novel candidates based on the input-setting combinations (isolation forest)
  - Union of the two models as prediction to maximize the bug capture
  - 80% UFS recovered by running about 60% of the tests.
- Deployment
  - Python application in HPC
  - Complementary flow
  - Daily batch simulation



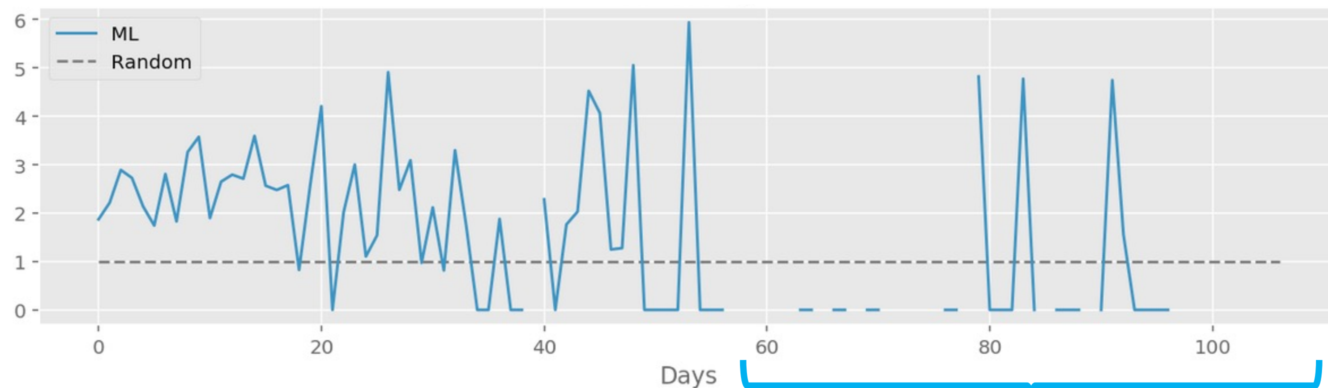
Post deployment treatment

- Continuous monitoring and periodic retraining (weekly)
- Optimizing the training window and applying recency effect
- Applying performance-weights to determine the size of candidate pools from each model
- Using ranking based on prediction probability instead of binary labels
- Preventing information leakage with time-sensitive cross-validation

Unique failure signature discovery rate (per test)



Efficiency (x times more efficient than the default)



- Tests were not run on most days.
- Model performance is less stable.

Challenges and future steps

Challenges	Future steps
Reduced stability in model performance towards the end of projects	• Automated feature engineering • Flexible and adaptive retraining • Surgical approach to target specific signatures
Stochasticity in test bench design + lack of systematic approach towards input settings	• Automated feature assessment • Counterfactuals
Debugging model performance especially regarding design and testbench changes	• Identify key input settings • Identify critical fail signatures • Quantify design and testbench changes
Efficient feedback loop between ML engineers and verification engineers	• Engineering platform to capture debugging process

Questions