Model-Driven Testing

A Property-Based Approach for End-to-End Testing

Nisan Haramati
@nisanharamati
nisan@haramati.ca

tldl; Model-Driven Testing - The Why

- Testing a distributed systems framework
- Test space too big

```
      { input                   }
    x { add/remove nodes        }
    x { crash/recover nodes     }
    x { application topologies  }
```

- End-to-End properties
- Reproducibility

tldl; Model-Driven Testing - The What

- From **End-to-End Testing**
  - Programmatic instrumentation
  - System as a gray/black box
- From **Property-Based Testing**
  - Fuzzing
  - Focus on properties
  - Broad specification

tldl; Model-Driven Testing - The What

- **Adding**
  - Model: validation context for state transitions
    - Is the new state reachable from the previous state?
  - Progressive validation
    - History-dependence
      - Is the new state valid, given the previous state(s)?

tldl; Example - Cassandra Cluster Size

- Operations: add_nodes, remove_nodes
- Properties
  - ```
    size( add([...]), cluster) ==
        size( cluster ) + size([...] )
    ```
  - ```
    size(remove([...]), cluster) ==
        size( cluster ) - size([...] )
    ```

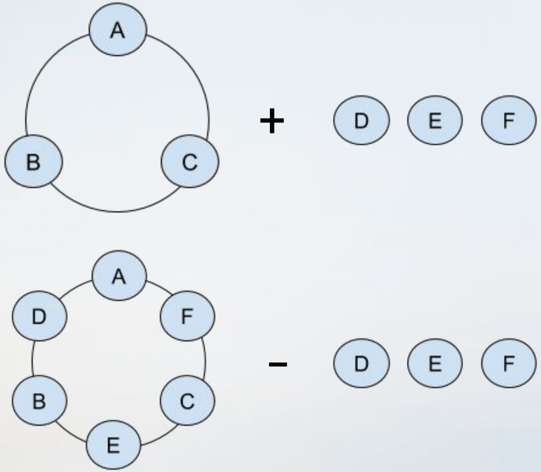
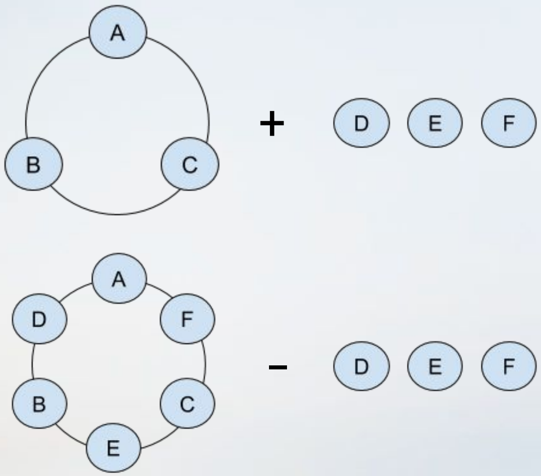


Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

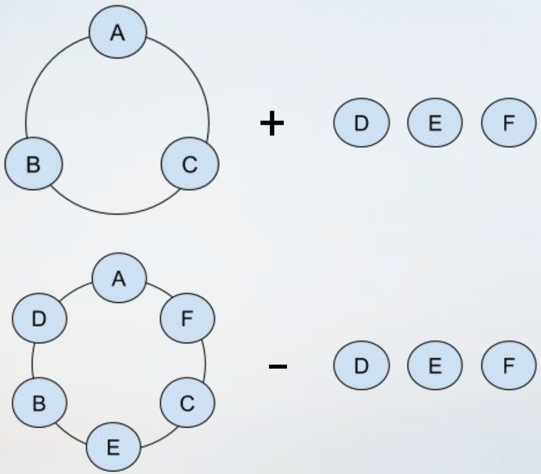
tldl; Example - Cassandra Cluster Size

- What can possibly go wrong?
  - Cluster too big (can't split data further)
  - Cluster overloaded (can't handle the overhead)
  - Degraded availability
  - Network partition
  - Full disk
  - Noisy neighbour
  - Bad configuration



tldl; Example - Cassandra Cluster Size

- What can possibly go wrong?
- Testing simple properties can reveal deeply hidden pathologies



tldl; Is this Model Based Testing?

- Short answer: Yes. Sort of.
  - **Key Concept:** Model as validation context
- Long answer: No. Sort of.
  - **Key Difference:** Model isn't restricted to test generation and output validation.
- Important distinction in distributed systems tests

Agenda

- Too long didn’t listen;
- Background
- The Challenge: Testing a Complex Distributed Framework
- Model-Driven Testing
- Examples
- Conclusions
- References

About me

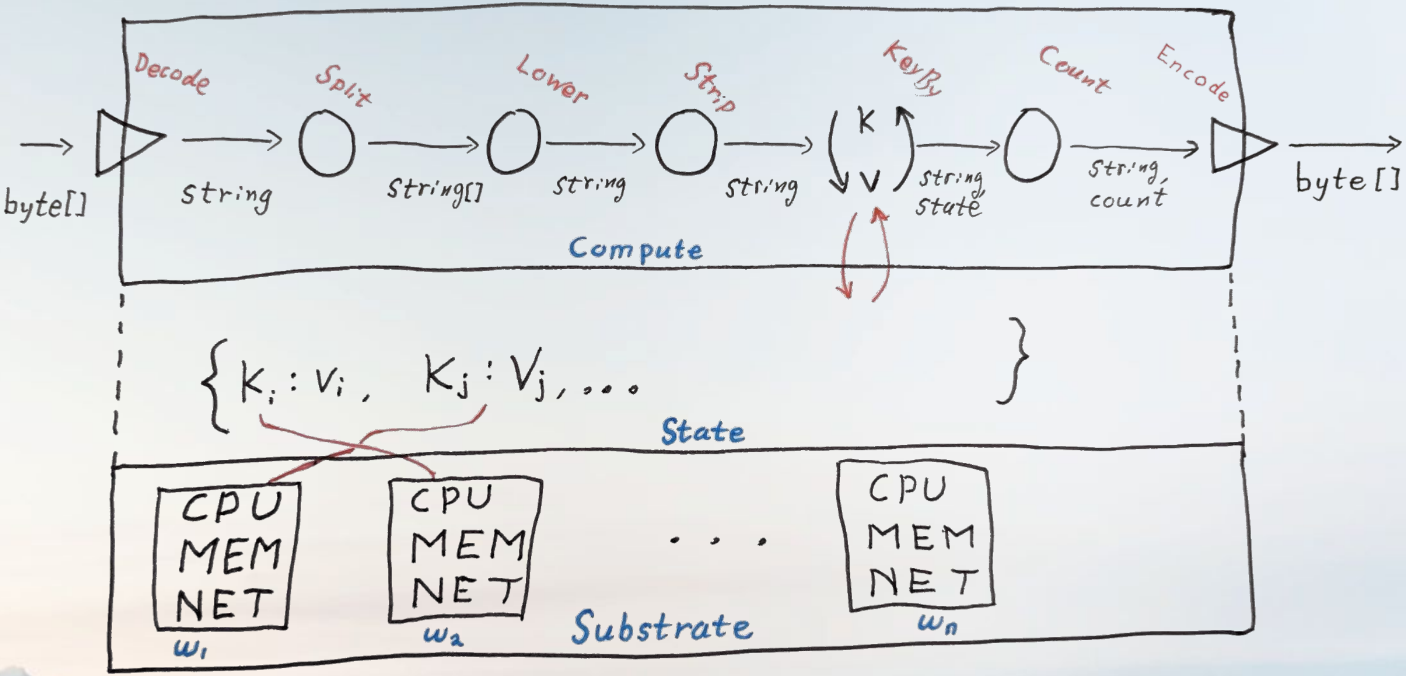
- Distributed Systems at Wallaroo Labs
- Real-time Complex Event Processing
- Data Quality in Real-time and Distributed Systems
- Data Engineering and Infrastructure
  - Online dating, bioinformatics, fintech

Wallaroo

- Framework for distributed data processing apps
  - Managed state
  - Application as a computation graph
  - Scale, concurrency, distribution, reliability
- Written in Ponylang
- Similar to Apache Flink

Word Count in Wallaroo

Multiple layers of abstraction



Word Count in Wallaroo

- ➢ Resource
- ➢ State
- ➢ Compute
- ➢ Application as Code

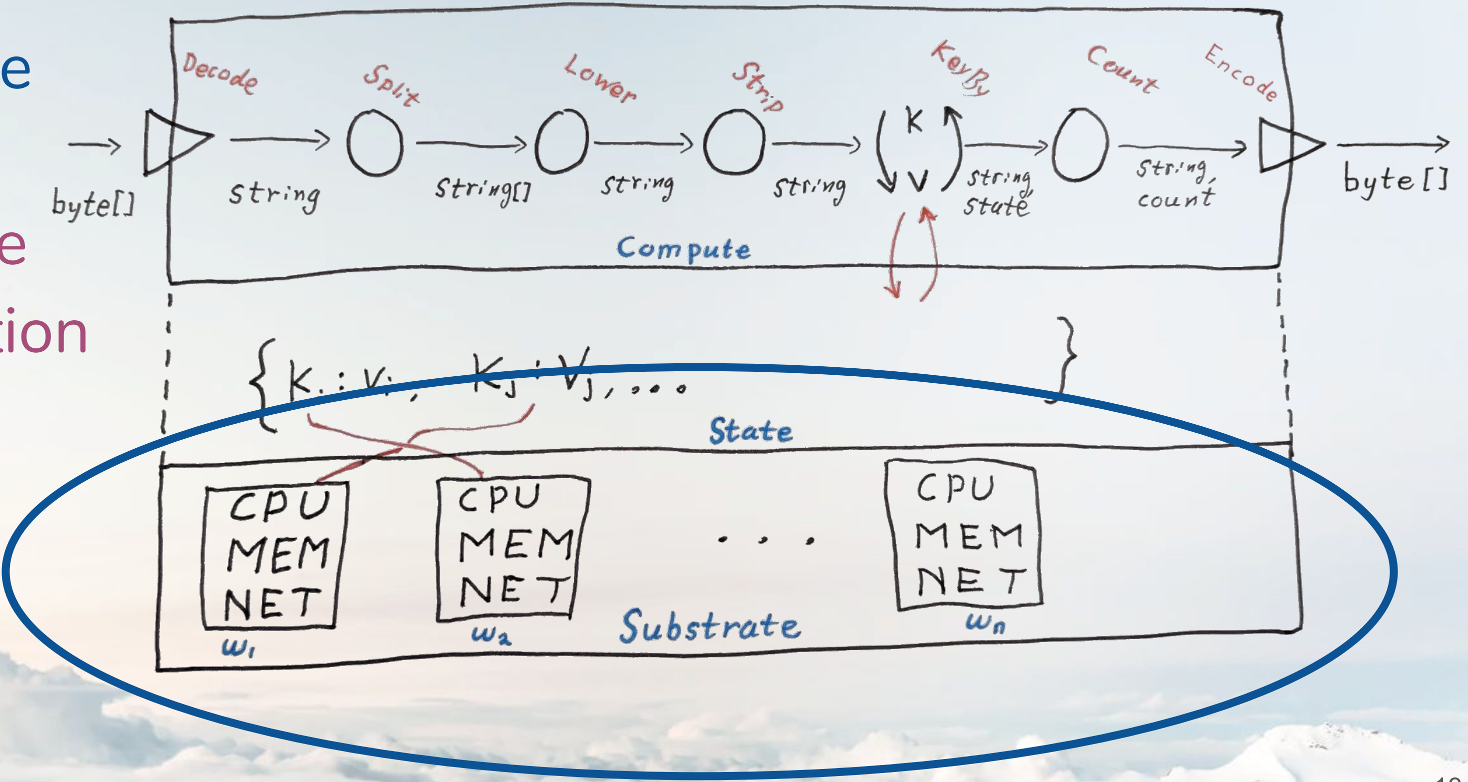
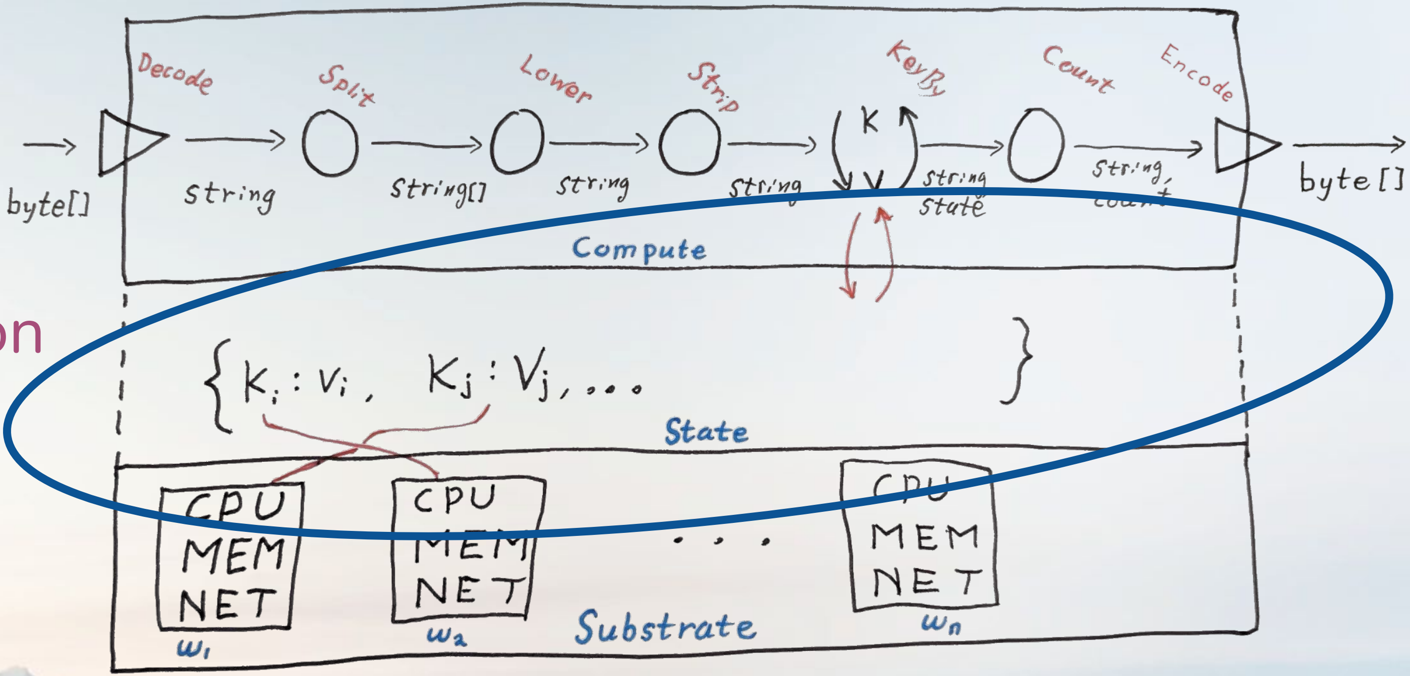


Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

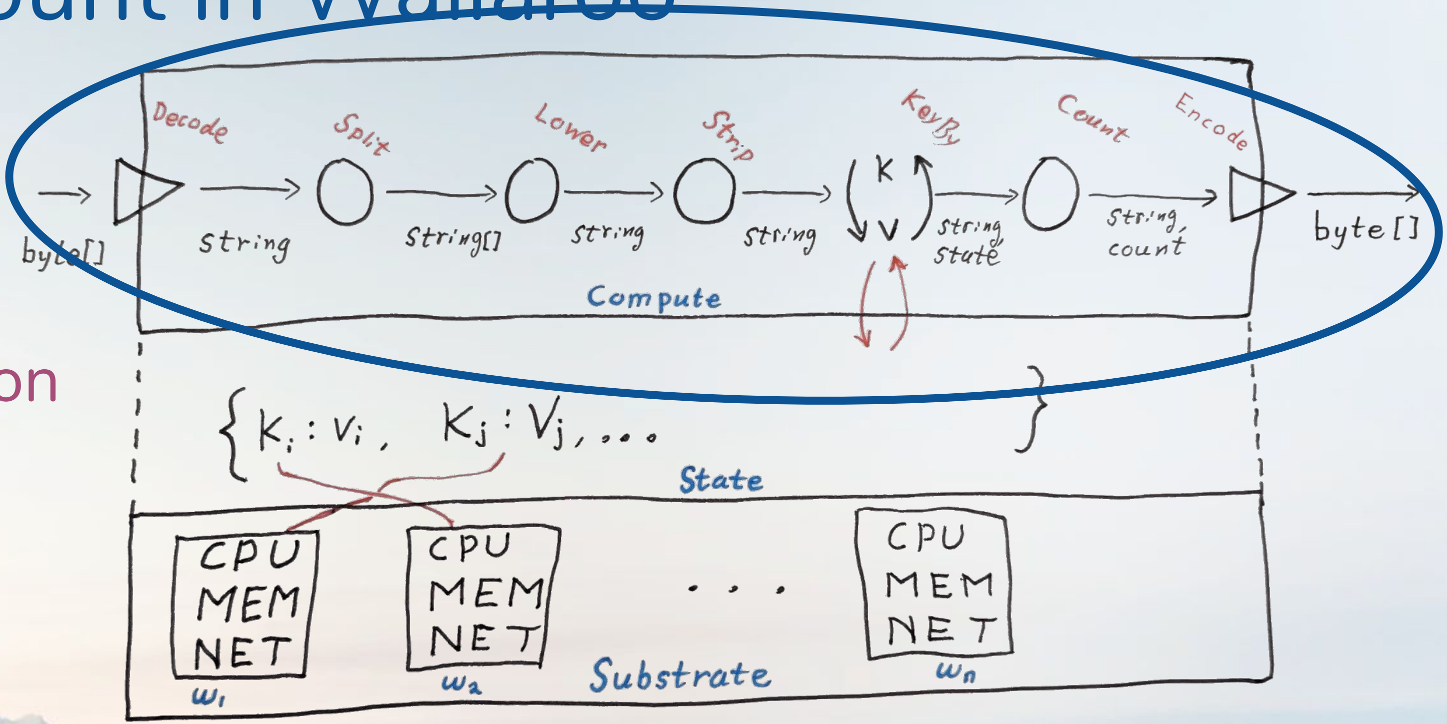
Word Count in Wallaroo

- ➢ Resource
- ➢ State
- ➢ Compute
- ➢ Application as Code



Word Count in Wallaroo

- ➢ Resource
- ➢ State
- ➢ Compute
- ➢ Application as Code



Word Count in Wallaroo

- Resource
- State
- Compute
- Topology as Code

```
Source(Decode)
    .to(Split)
    .to(Lower)
    .to(Strip)
    .key_by(MyKeyFunction)
    .to(Count)
    .to_sink(Encode)
```

The Challenge:

Testing a Complex Distributed Framework

Wallaroo Characteristics

- ➢ Distributed → Orchestration
- ➢ Real-time → External dependencies (sources, sinks)
  → History dependence
- ➢ Opaque state → Signal generation
- ➢ Framework → Not directly testable
  → Large space of possible applications

Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

We Might Want to Test...

- Functional
  - `Output == Expectation(Input)`
- Operational
  - *Actually works*
  - Scales → Can add/remove workers
  - Reliable → Can recover from worker failure

We Might Want to Test...

- Qualitative → Core Guarantees
  - Consistency (causal)
  - Everything arrives
    - Where it should
    - In order
    - Without loss or duplication

Model-Driven Testing

Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

Property-based testing

Is this a unicorn?

- Has 1 horn
- Has 4 legs
- Has 1 tail
- Has 2 ears

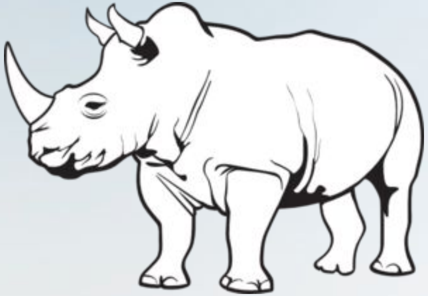
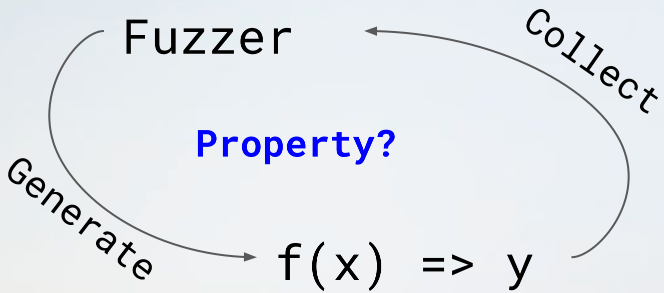


Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

Property-based testing

1. A fuzzer
2. A library of tools for making it easy to construct property-based tests using that fuzzer.

- Dr. MacIver, hypothesis.works



Fuzzer

- Produce input data for the test
- Possibly dynamically generated
- Possibly dependent on results of previous runs

  - Dr. MacIver, hypothesis.works

Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

Property-based testing

```
def sum(num1, num2):
  """Return the sum of two numbers"""
  return num1 + num2

# Unit test
def test_unit_sum():
  assert(sum(1,2) == 3)
```

Property-based testing

```python
def sum(num1, num2):
  """Return the sum of two numbers"""
  return num1 + num2 if num2 < 500000 else 0

# Property Based test
def test_property_sum():
    # fuzz loop
    from random import randrange
    # generate a million random pairs
    for _ in range(1000000):
        n1 = randrange(-1000000000, 1000000000)
        n2 = randrange(-1000000000, 1000000000)

        # Test the sum property
        assert( sum(n1, n2) == n1 + n2)
```

Property-based testing

```
    def test_property_sum():
        # fuzz loop
        from random import randrange
        # generate a million random pairs
        for _ in range(1000000):
            n1 = randrange(-1000000000, 1000000000)
            n2 = randrange(-1000000000, 1000000000)

            # Test the sum property
>           assert( sum(n1, n2) == n1 + n2)
E           assert 0 == (476988046 + 25202221)
E            +  where 0 = sum(476988046, 25202221)

test_sum.py:22: AssertionError
```

End-to-End Properties

- Functional Correctness
- Operational Acceptance
  - Robustness, reliability
- Qualitative Correctness
  - Consistency

The End-to-End Problem

- Wallaroo is not a pure function… or a class… or even a single executable
- Need
  - Orchestration
  - Remote control and measurement
    - A distributed systems problem
    - Order of concurrent events, clock skew, asynchronous

The End-to-End Problem

- For every single test
  - **Start** Wallaroo cluster, sinks, sources
  - Get it into a **specific state**
  - Send **input**, induce an **event**, or inject a **fault**
  - **Measure** before, during, after

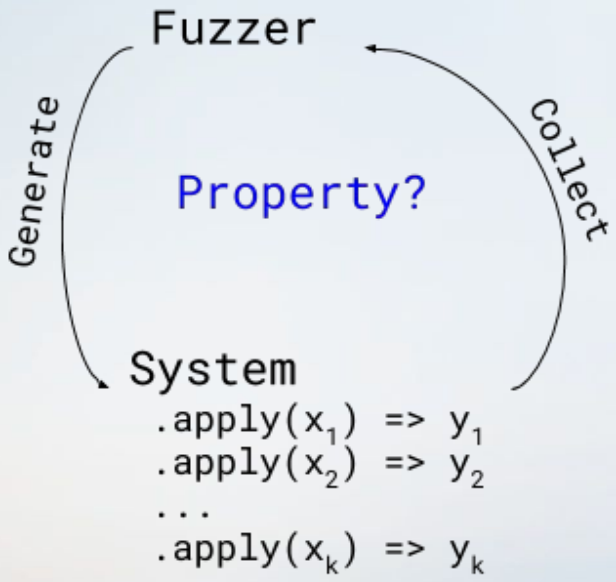


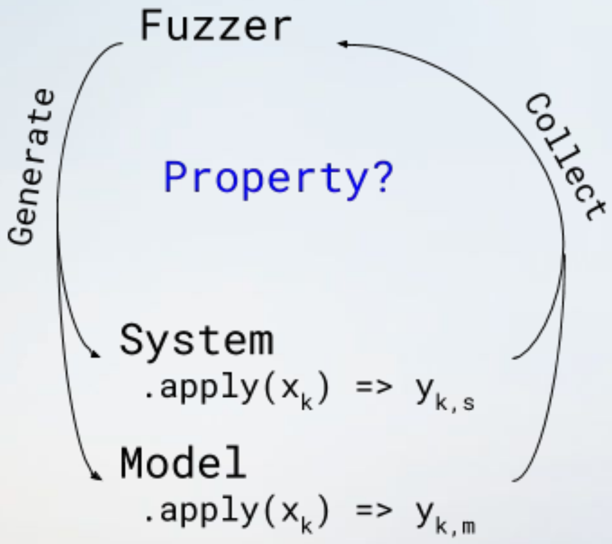
Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

Wallaroo - End-to-End Testing

```
# Start a cluster
with Cluster(command=?,host=?,
             sources=?,workers=?,
             sinks=?,sink_mode=?,
             ...) as cluster:
  # Start source streams
  …
  # Execute test events
  …
```

Model-Based Testing

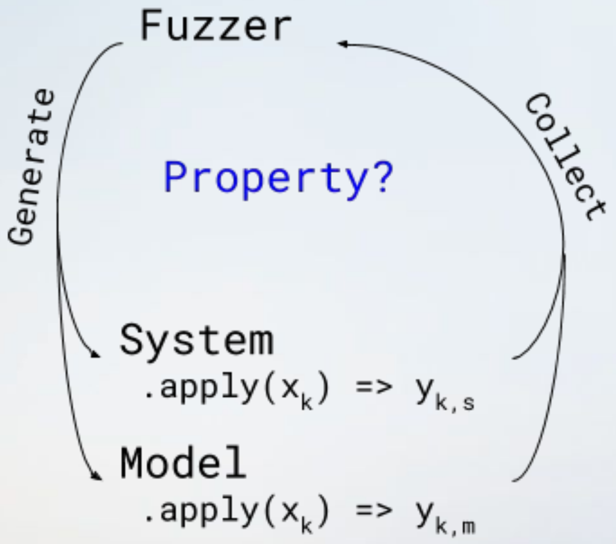
- PBT+ E2E + **Model**
- Model informs
  - Input generator
  - Event generator
  - Fault generator
  - Online/offline validation
- Generators may try to cover state space



Model-Based Testing

- Events are applied to
  - a **system** under test
  - a **model** of the system

    properties as states (e.g. an FSM)
- After each application, the properties of the SUT and the model are compared



Model-Driven Testing

- Events are applied to
  - a **distributed system** under test
  - a **model** of the system properties as states (e.g. an FSM)
- After each application…

measurement may not be possible

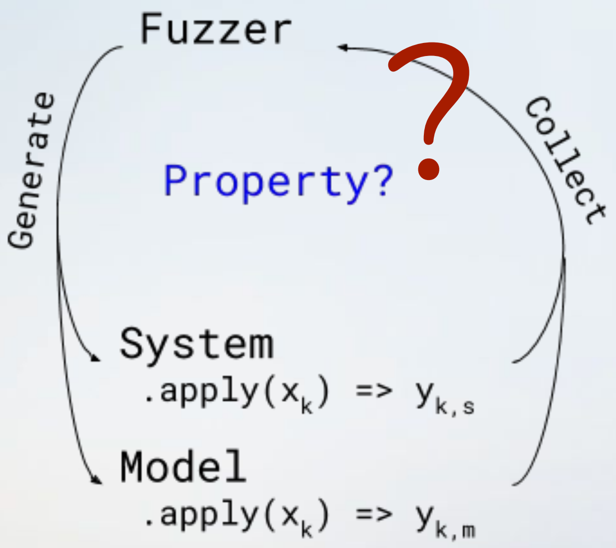
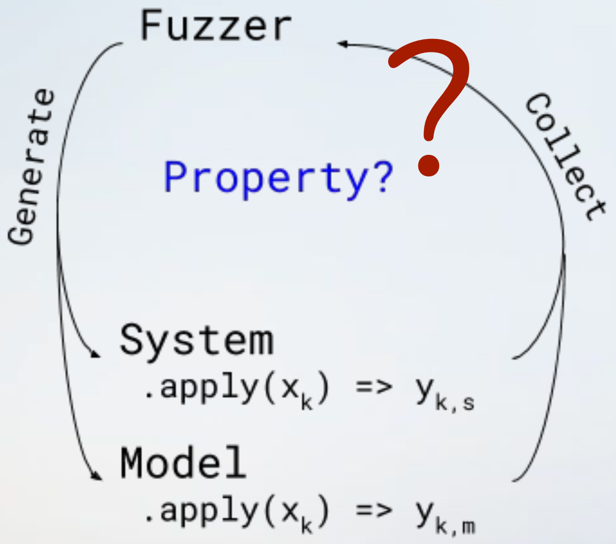


Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed

Model-Driven Testing

- Signal & Measurement
- Self-validating applications
  - Can we validate guarantees within the test application?
  - The `.apply(...)` may include validation logic



Examples

Signal & Measurement

- Properties
  - Ordered
  - No loss
  - No duplication
- State is Opaque

- Operations
  - Scaling
    - add / remove nodes
  - Reliability
    - crash / recover nodes

State Consistency Signal

```
Count(word, total) ⇒
  total += 1
  return total
```

$(op_0, state_0) \rightarrow (op_1, state_1)$

Op	State before	**Output**
0	0	**1**
1	1	**2**
2	2	**3**
3	3	**4**
…	…	**…**
n	n	**n+1**

State Consistency Signal

```
Count(word, history) ⇒
  new_count = history.last + 1
  history.push(new_count)
  return history

(Count_1("dog"), [0]) →
(Count_2("dog"), [0,1,]) → …
(Count_n("dog"), [0,1,…,n-1,n])
```

Op	State Before	**Output**
0	[0]	**[0,1]**
1	[0,1]	**[0,1,2]**
2	[0,1,2]	**[0,1,2,3]**
3	[0,1,2,3]	**[0,1,2,3,4]**
…	…	**…**
n	[…,n-1,n]	**[…,n,n+1]**

State Consistency Signal

```
(Count_1("dog"), [0]) →
(Count_2("dog"), [0,1]) → …
(Count_50("dog"), [...,48,49]) →
        >>CRASH>>
        <<RECOVER...
          ROLLBACK<<
(Count_51("dog"), ???)
              [...,49,50] + [51] ✓
              [...,41,42] + [51] ✗
```

Op	State	Output
...	...	...
50	[...,48,49]	**[...,49,50]**
…		
51 (good)	[...,49,50]	**[...,50,51]**
51 (bad)	[...,41,42]	**[...,42,51]**

Inconsistent State Detection

At the output (offline validation):

```
[...,49,50,51] ✓
[...,41,42,51] ✗
```

Op	State	Output
...	...	...
50	[...,48,49]	**[...,49,50]**
...		
51 (good)	[...,49,50]	**[...,50,51]**
51 (bad)	[...,41,42]	**[...,42,51]**

Inconsistent State Detection

On update (online validation):

- `+1 logic` is insufficient
- Need sequence info of input message

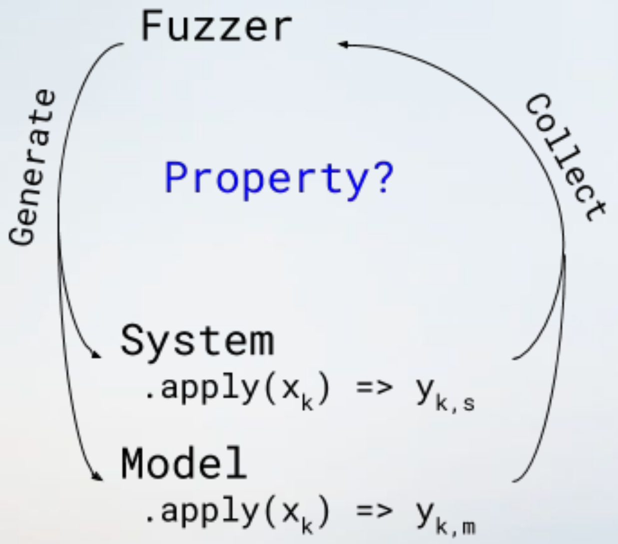
```
Observe(next, history) ⇒
  if next != history.last +1:
    crash("Sequentiality error!")
  history.push(next)
  return history
```

Op	State	Next	Next - last
...	...		...
50	[...,48,49]	50	1
...			
51 (good)	[...,49,50]	51	1
51 (bad)	[...,41,42]	51	9

Wallaroo - Scaling and Recovery Tests

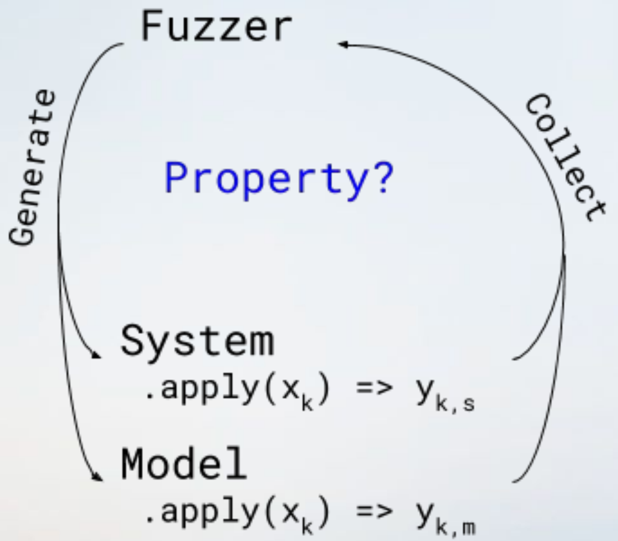
```
# For a given set of operations
ops = [Grow(2), Shrink(1),Crash(2)
       Recover(2),Grow(1)]

# Start a cluster
with Cluster(...) as cluster:
  # Start source streams
  ...
  # Execute test operations
  for event in ops:
    event.apply(cluster)
```



Wallaroo - Scaling and Recovery Tests

```
# Dense matrix test generator
for api, group in APIS.items():
  for app in group:
    for ops in SEQS:
      for src_type in SOURCE_TYPES:
        # Create & execute tests
        ...
        # 30 Recovery test sequences
        # 144 Scaling test sequences
```



Wallaroo - Topology Tests

- Recall word count
- Application topologies are user-defined
  - Infinitely many
  - Like testing a VM or a compiler

```
Source(Decode)
    .to(Split)
    .to(Lower)
    .to(Strip)
    .key_by(MyKeyFunction)
    .to(Count)
    .to_sink(Encode)
```

Wallaroo - Topology Tests

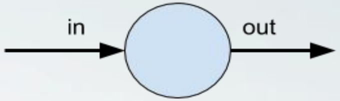
- Recall word count
- Application topologies are user-defined
- How can we test this?
- **Code generation**

```
Source(Decode)
    .to(Split)
    .to(Lower)
    .to(Strip)
    .key_by(MyKeyFunction)
    .to(Count)
    .to_sink(Encode)
```

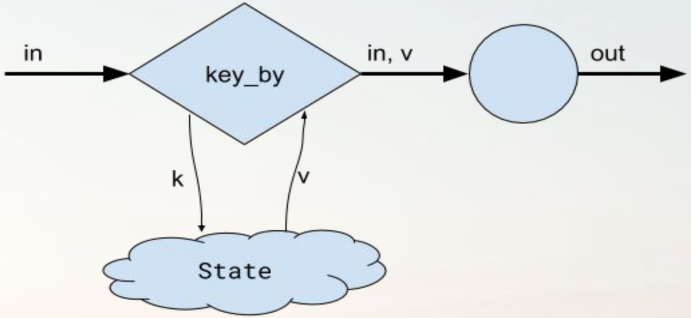
Wallaroo - Topology Instrinsics

- Computations

Stateless

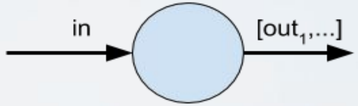


State + routing

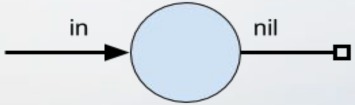


- Concurrency
- Flow Modifiers

One-to-many



Filter



Wallaroo - Generative Topology Tests

- Intrinsics → basis
- Test cross product of

```
    { computations   }
x   { concurrency    }
x   { flow modifiers }
x   { cluster size   }
```

Wallaroo - Generative Topology Tests

- Intrinsics → basis
- Test cross product of

```
    { computations    }
x   { concurrency     }
x   { flow modifiers  }
x   { topology depth  }
x   { cluster size    }
```

- Tracer app
  - Append step ID and monotonic counter value
  - Send message forward
- Validation
  - Reconstruct topology from trace output
  - Compare against known application topology

Wallaroo - Generative Topology Tests

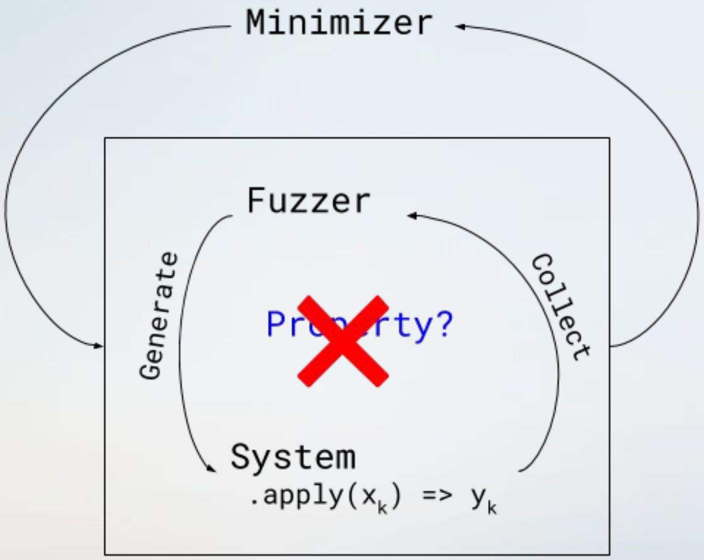
- Intrinsics → basis
- Test cross product of

```
    { computations    }
x   { concurrency     }
x   { flow modifiers  }
x   { topology depth  }
x   { cluster size    }
```

```
# Create topology sequences
for d in depths:
  for steps in product(groups, d):
    for size in cluster_sizes:
      # Create & execute tests
      …
      # Process output traces and
      # match against ‘steps’
      …
# 504 Topology tests
```

Refinement and Minimization

- ➢ After we find a failing test case
  - ➢ Alert and stop
  - ➢ Try to minimize test input
    - ➢ Easy* for 1-dimensional fuzzer
    - ➢ Model dependent for multi-dimensional fuzzer



* still difficult

In Summary - Model-Driven Testing

- ➢ Property-Based Tests for End-to-End Properties
  - ➢ Functional, **Operational**, and **Qualitative** properties
  - ➢ Distributed systems testing
  - ➢ When measurements are hard or impossible
- ➢ Another layer on top of unit, integration, system, and end-to-end testing

In Summary - Model-Driven Testing

- Requires
  - End-to-End instrumentation (provision, deploy, run, control, collect, teardown)
  - A model of the properties being tested
  - A test generator
- Reduces work required to cover a large test space

References

- Hillel Wayne on Types of tests: https://www.hillelwayne.com/post/a-bunch-of-tests/
- Model Based Testing - https://en.wikipedia.org/wiki/Model-based_testing
- Hypothesis, Property-based testing for Python - https://hypothesis.works/
- Testing a Distributed System - https://queue.acm.org/detail.cfm?id=2800697
- Wallaroo - https://github.com/WallarooLabs/wallaroo

Thank you!

Nisan Haramati
@nisanharamati
haramati.ca
nisan@haramati.ca

Photo credit: Dominik Schröder, pub.ist.ac.at/~dschroed