

Simulation Testing

@mtnygard



cognitect

Classification

classification

Example-Based

Property-Based

White Box

Black Box

Automated

Manual

Example-Based Testing

example-based testing

Programmer probes the code

Programmer creates test cases

Validate expectations against outcomes

example-based testing

```
(are [x y] (= x y))
```

```
(+ ) 0
```

```
(+ 1) 1
```

```
(+ 1 2) 3
```

```
(+ 1 2 3) 6
```

```
(+ -1) -1
```

```
(+ -1 -2) -3
```

```
(+ -1 +2 -3) -2
```

```
(+ 2/3) 2/3
```

```
(+ 2/3 1) 5/3
```

```
(+ 2/3 1/3) 1 )
```

example-based testing

(are [x y] (= x y))

(+) 0

(+ 1) 1

(+ 1 2) 3

(+ 1 2 3) 6

(+ -1) -1

(+ -1 -2) -3

(+ -1 +2 -3) -2

(+ 2/3) 2/3

(+ 2/3 1) 5/3

(+ 2/3 1/3) 1)

no setup

example-based testing

(are [x y] (= x y))

(+) 0

(+ 1) 1

(+ 1 2) 3

(+ 1 2 3) 6

(+ -1) -1

(+ -1 -2) -3

(+ -1 +2 -3) -2

(+ 2/3) 2/3

(+ 2/3 1) 5/3

(+ 2/3 1/3) 1)

inputs

example-based testing

(are [x y] (= x y))

(+) 0

(+ 1) 1

(+ 1 2) 3

(+ 1 2 3) 6

(+ -1) -1

(+ -1 -2) -3

(+ -1 +2 -3) -2

(+ 2/3) 2/3

(+ 2/3 1) 5/3

(+ 2/3 1/3) 1)

execution



example-based testing

```
(are [x y] (= x y))  
  (+) 0  
  (+ 1) 1  
  (+ 1 2) 3  
  (+ 1 2 3) 6  
  
  (+ -1) -1  
  (+ -1 -2) -3  
  (+ -1 +2 -3) -2  
  
  (+ 2/3) 2/3  
  (+ 2/3 1) 5/3  
  (+ 2/3 1/3) 1 )
```



outputs

example-based testing

(are [x y] (= x y))

validation

(+) 0

(+ 1) 1

(+ 1 2) 3

(+ 1 2 3) 6

(+ -1) -1

(+ -1 -2) -3

(+ -1 +2 -3) -2

(+ 2/3) 2/3

(+ 2/3 1) 5/3

(+ 2/3 1/3) 1)

examples of examples

Scales: Unit, Functional, Acceptance

Styles: Test-After, TDD, BDD

Common Idioms: Fixtures, Stubs, Mocks

weaknesses of examples

severely limited coverage

fragility

poor scalability

non-composability

Property-Based Testing

property-based testing

Programmer models the *domain* and *invariants*

A *program* generates many individual tests


Validates categoric properties, not specific outcomes

test.check

<https://github.com/clojure/test.check>

property example

property name



```
(def sort-preserves-length
  (prop/for-all [v (gen/vector gen/int)]
    (let [s (sort v)]
      (= (count v) (count s)))))
```

property example

quantifier



```
(def sort-preserves-length
  (prop/for-all [v (gen/vector gen/int)]
    (let [s (sort v)]
      (= (count v) (count s)))))
```

property example

```
(def sort-preserves-length  
  (prop/for-all [v (gen/vector gen/int)]  
    (let [s (sort v)]  
      (= (count v) (count s)))))
```



input generator

property example

```
(def sort-preserves-length  
  (prop/for-all [v (gen/vector gen/int)]  
    (let [s (sort v)]  
      (= (count v) (count s)))))
```



fn under test

property example

```
(def sort-preserves-length  
  (prop/for-all [v (gen/vector gen/int)]  
    (let [s (sort v)]  
      (= (count v) (count s)))))
```



validation

Simulation Testing

simulation testing

Example-Based

Property-Based

White Box

Black Box

Automated

Manual

simulation testing

Example-Based

Property-Based

White Box

Black Box

Automated

Manual

simulation testing

Extend property-based testing to whole systems

Programmer models the *domain*

Program generates a *repeatable script* of inputs

System tested in-situ

Validate categoric and global properties repeatedly

what is testing?

attempts to prove the system wrong

not an attempt to prove the system is right

search problem: find interactions that result in forbidden zone of state space

coverage

code coverage is misleading

state space coverage is better

trajectory coverage is better still

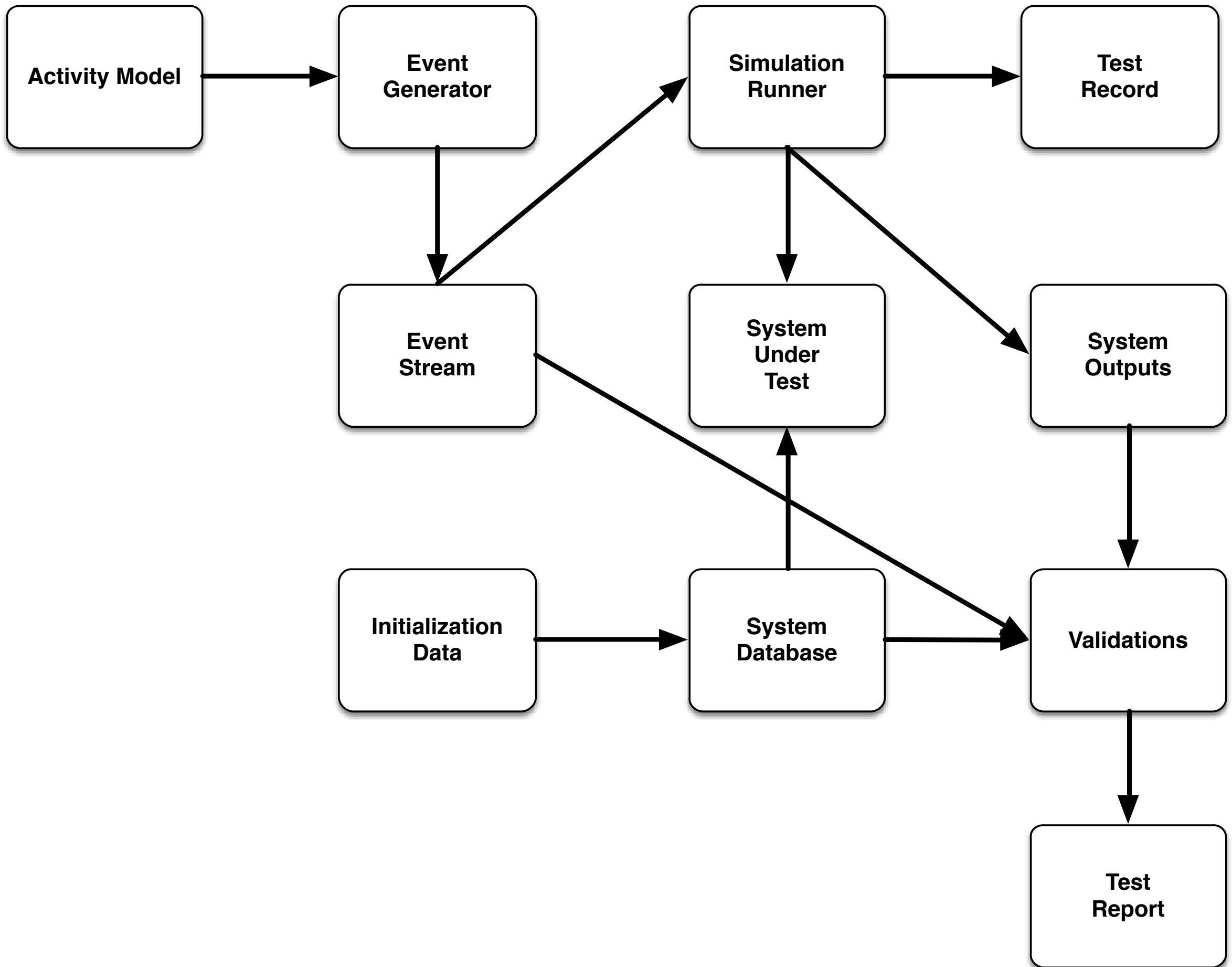
coverage

we must put bounds on test execution time

but want the greatest confidence for that time

hence:

exercise many, many trajectories



so many moving parts ?!

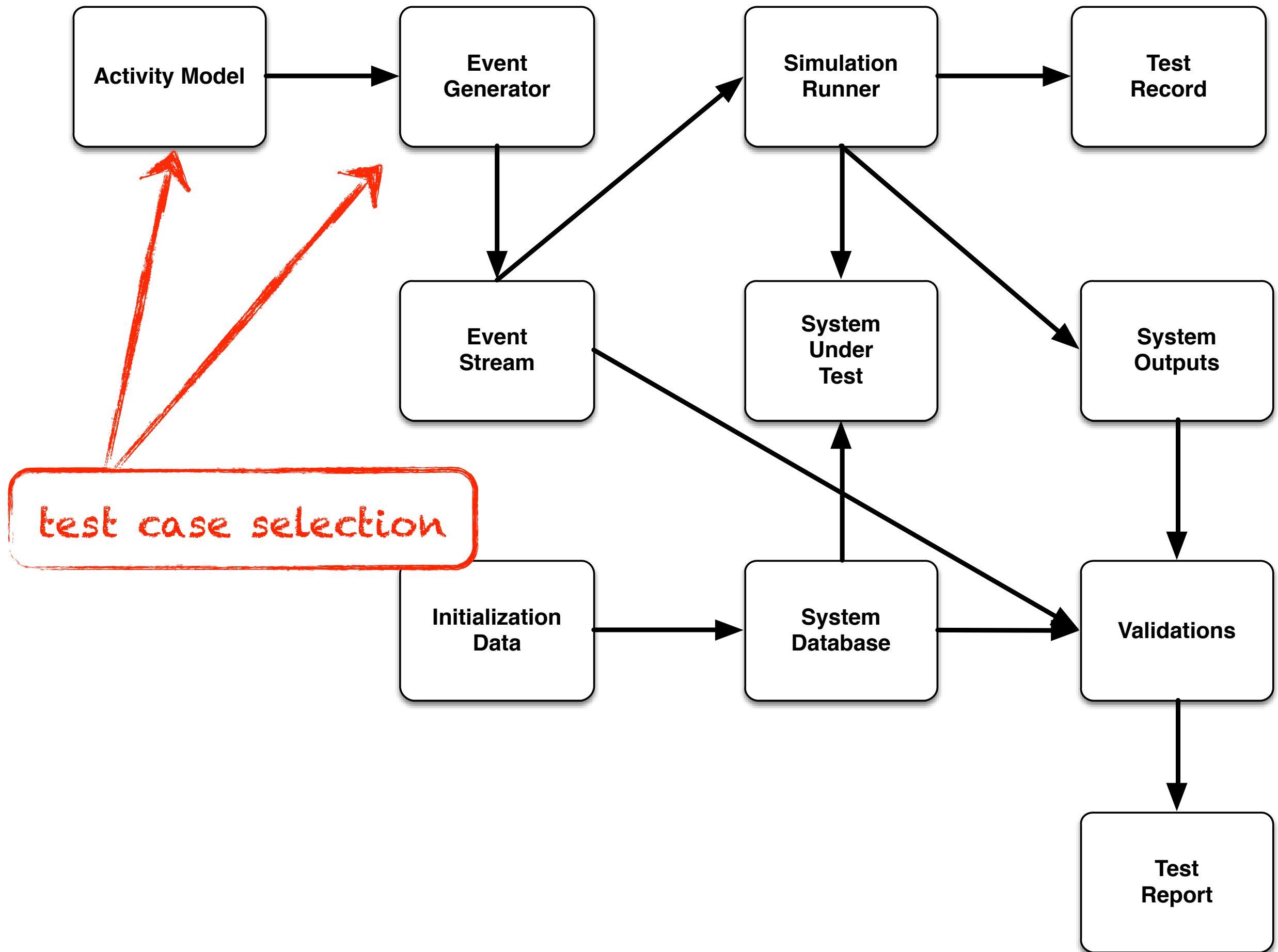
decoupling via immutable databases

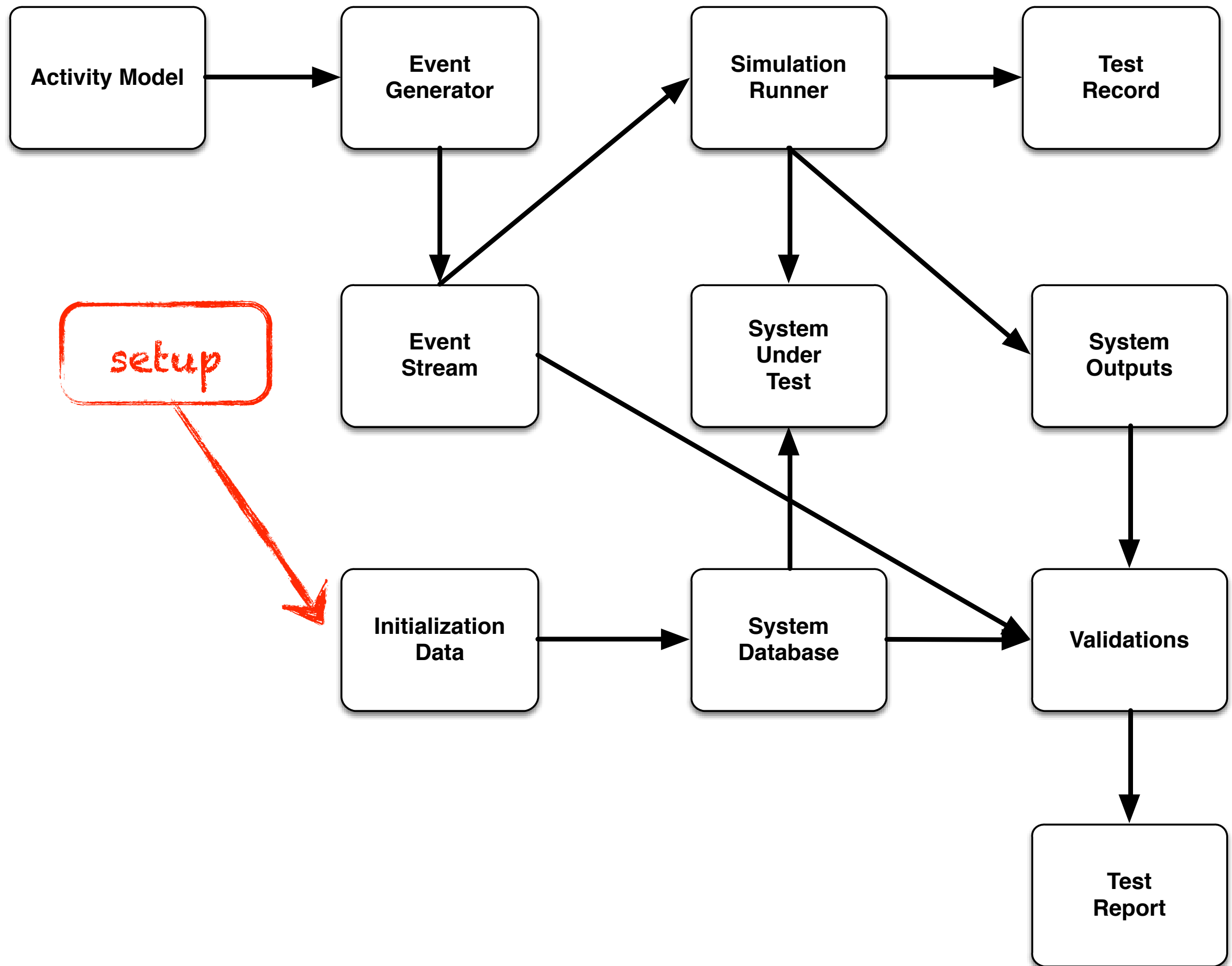
repeatable tests

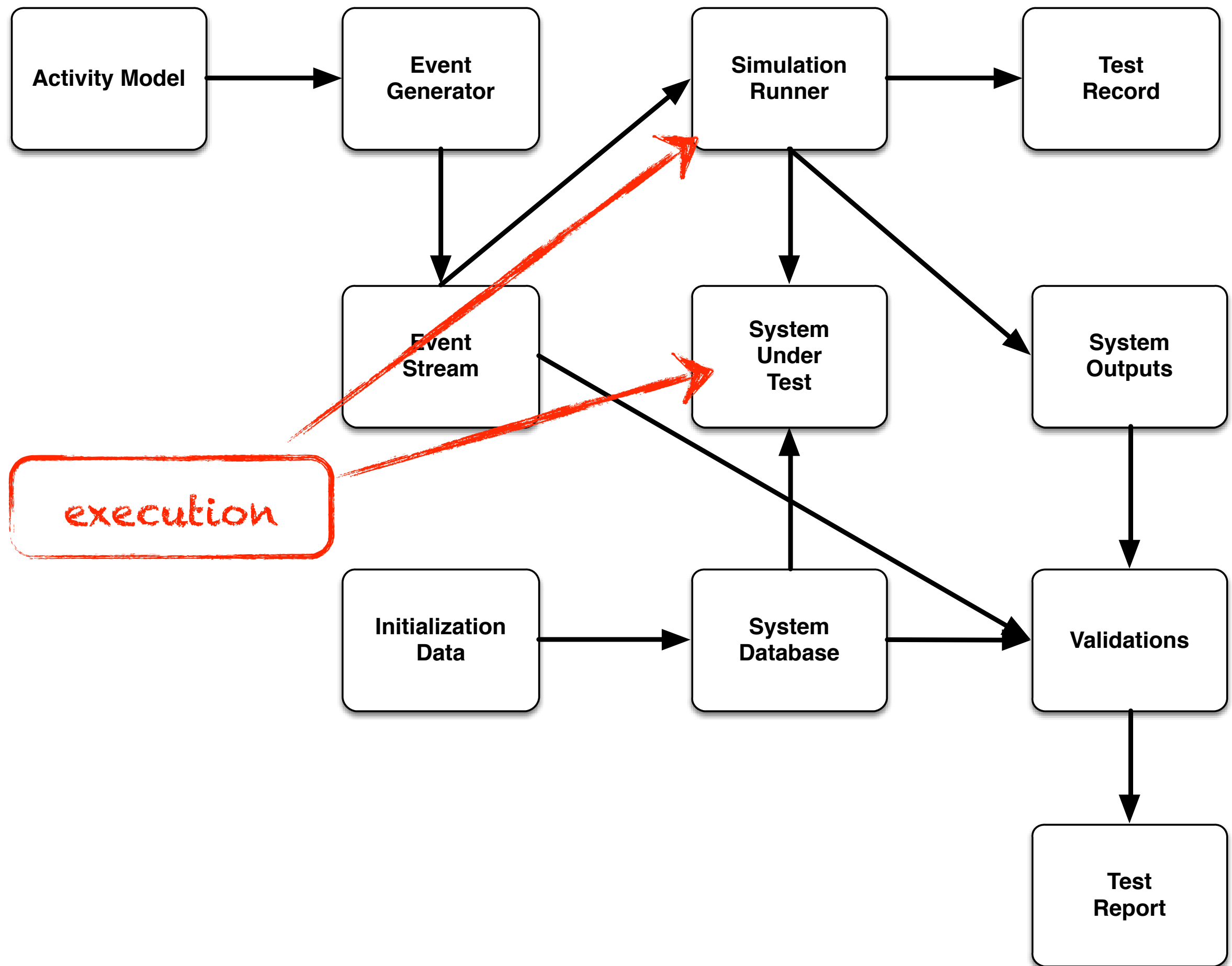
repeatable verifications

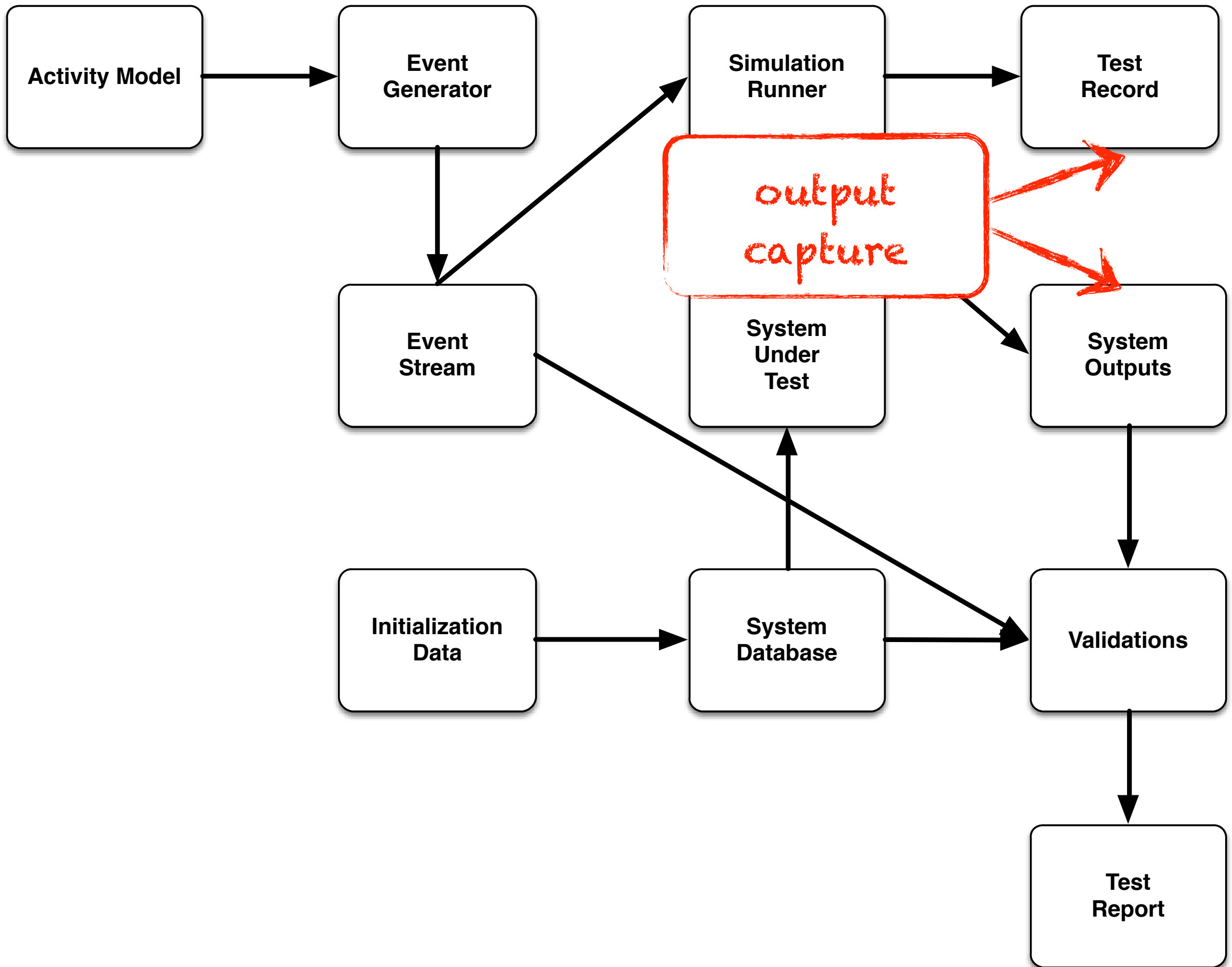
Parts of Every Test

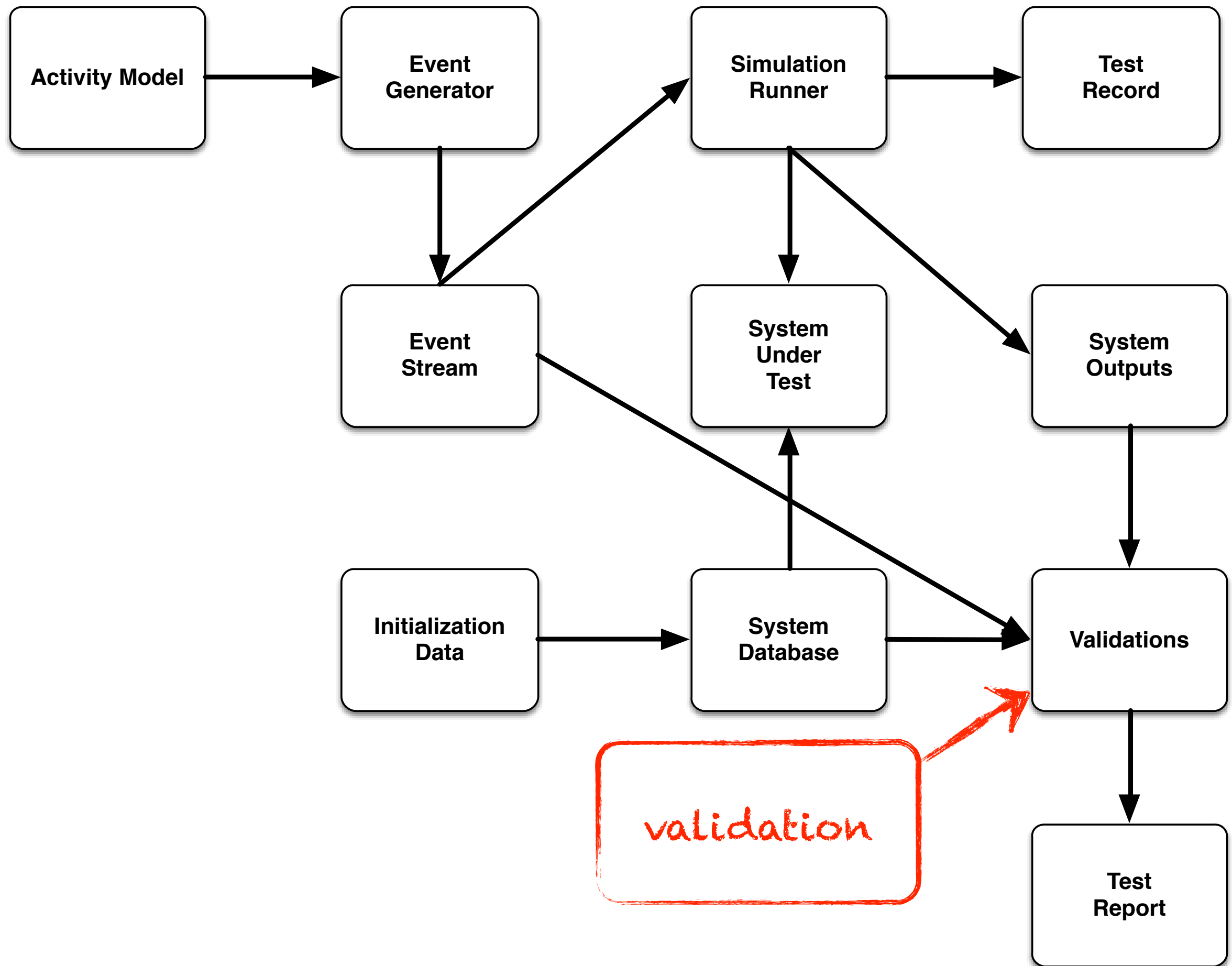
1. Test case selection
2. Setup
3. Execution
4. Output capture
5. Validation
6. Report

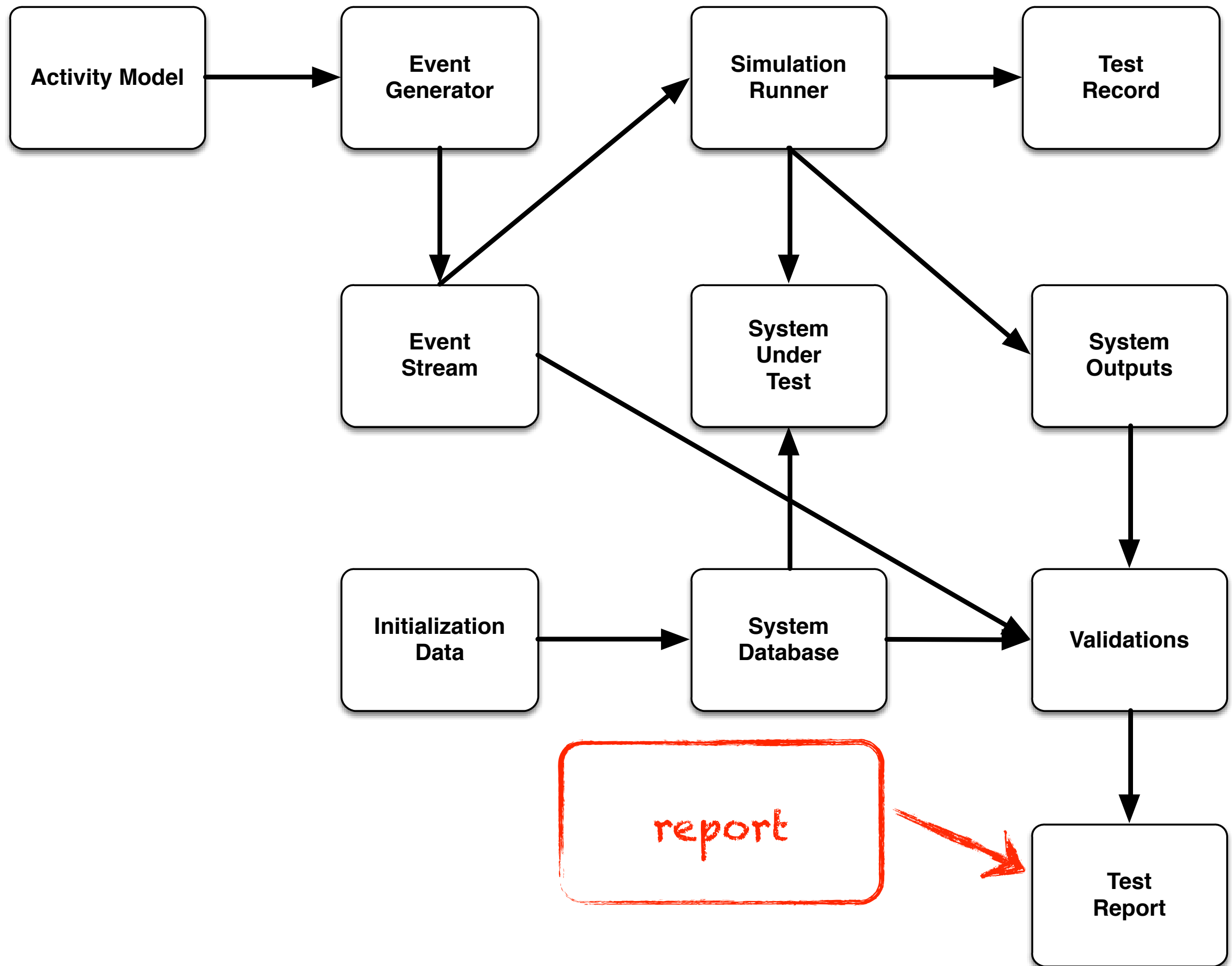












Simulant

Open source framework for simulation testing

<https://github.com/Datomic/simulant>

model

small set of parameters

depends on the nature of your system

model example

```
(defn edges
  [multi-session-odds]
  ;; start-state          end-state          weight  max-delay
  [[:start               :token-acquired      100      10]
   [:token-acquired      :idle                100      10]
   , , ,
   [:purchase-made       :idle                10      5000]
   [:purchase-made       :session-end         90      5000]
   [:session-end         :start               multi-session-odds 5000]
   [:session-end         :halt (- 100 multi-session-odds) 5000]])
```


model example

```
(defn state-sequence [model test]
  (es/event-stream
    (state-transition-model
      (:model/multiDeviceUserSessionPercentage model))
    [(initial-state test)])))
```

<https://github.com/candera/causatum>

generator

creates an unlimited amount of activity

contains all the randomness

everything later is deterministic*

generator example - actions

```
(defmulti actions-on-entry (fn [_ _ state] (:state state)))
```

```
(defmethod actions-on-entry :start  
  [model agent state]  
  [(action agent state :action.type/startSession)])
```

```
(defmethod actions-on-entry :idle  
  [model agent state]  
  [])
```

```
(defmethod actions-on-entry :session-end  
  [model agent state]  
  [(action agent state :action.type/endSession)])
```

generator example - actions

```
(defn- generate-actions
  [model test agents]
  (flatten
    (for [agent agents
          state (m/state-sequence model test)]
      (actions-on-entry model agent state))))
```

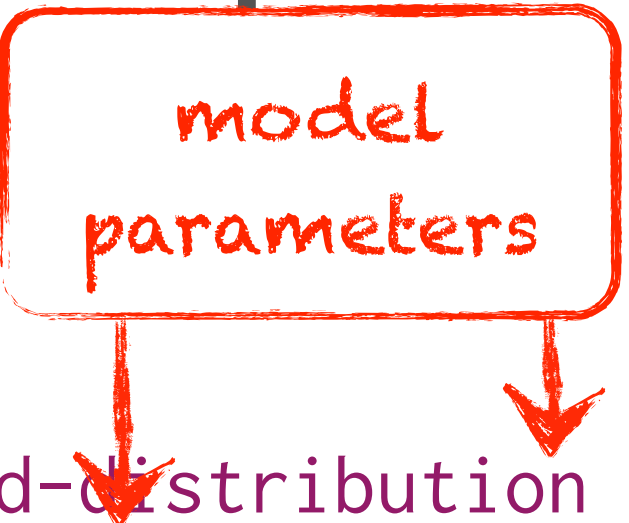
```
(defn- action [agent state type & {:as extra}]
  (merge
    {:db/id          (d/tempid :test)
     :agent/_actions (e agent)
     :action/atTime  (long (:rtime state))
     :action/type    type}
    extra))
```

generator example - agent

```
(defn- generate-agents
  [model test]
  (let [id-distr      (device-id-distribution model)
        likely-to-id (:model/identifyEarlyPercentage model)]
    (map
      (fn [db-id]
        { :db/id          db-id
          :agent/type     :agent.type/identifiedVisitor
          :agent/deviceIds (pick id-distr 3)
          :agent/userEmail (email-address)
          :agent/identifyEarly (> likely-to-id (rand 100))
          :test/_agents    (e test) })
      (repeatedly (:test/visitorCount test)
                  #(d/tempid :test))))))
```

generator example - agent

model
parameters



```
(defn- generate-agents
  [model test]
  (let [id-distr      (device-id-distribution model)
        likely-to-id (:model/identifyEarlyPercentage model)]
    (map
      (fn [db-id]
        { :db/id          db-id
          :agent/type     :agent.type/identifiedVisitor
          :agent/deviceIds (pick id-distr 3)
          :agent/userEmail (email-address)
          :agent/identifyEarly (> likely-to-id (rand 100))
          :test/_agents    (e test) })
      (repeatedly (:test/visitorCount test)
                  #(d/tempid :test))))))
```

generator example - agent

```
(defn- generate-agents
  [model test]
  (let [id-distr      (device-id-distribution n)
        likely-to-id (:model/identifyEarlyPerco
                      (device-id-distribution n))]
    (map
      (fn [db-id]
        { :db/id
          :agent/type
          :agent/deviceIds
          :agent/userEmail
          :agent/identifyEarly
          :test/_agents
          db-id
          :agent.type/identifiedVisitor
          (pick id-distr 3)
          (email-address)
          (> likely-to-id (rand 100))
          (e test)})
        (repeatedly (:test/visitorCount test)
                    #(d/tempid :test))))))
```

randomness
per agent

generator example - agent

```
(defn- generate-agents
  [model test]
  (let [id-distr      (device-id-distribution model)
        likely-to-id (:model/identifyEarlyPercentage model)]
    (map
      (fn [db-id]
        { :db/id          db-id
          :agent/type     :agent.type/identifiedVisitor
          :agent/deviceIds (pick id-distr 3)
          :agent/userEmail (email-address)
          :agent/identifyEarly (> likely-to-id (rand 100))
          :test/_agents     (e test) })
      (repeatedly (:test/visitorCount test)
                  #(d/tempid :test))))))
```



List of agents

simulation runner

sets up the system

executes the activities

captures external outputs

captures internal state as needed

runner example

```
(defn start-processes
  [{:keys [uri sim] :as ctx}]
  (assoc ctx
    :processes (->> #(sim/run-sim-process uri (:db/id sim))
                     (repeatedly (:sim/processCount sim))
                     (into []))))

(defn wait-for-completion
  [{:keys [processes] :as ctx}]
  (mapv (fn [p] @(:runner p)) processes)
  ctx)
```

runner example - lifecycle

```
(defn execute-sim
  [uri test-name process-count clock-multiplier]
  (-> {:uri uri
        :conn (d/connect uri)
        :test-name test-name
        :process-count process-count
        :clock-multiplier clock-multiplier}
    locate-test
    target-system/setup
    initialize-actions
    build-sim
    start-processes
    wait-for-completion
    retrieve-reports))
```

actions

repeatable

atomic step

no validation

limited error checking

action - example

```
(defmethod sim/perform-action :action.type/getUserToken
  [action process]
  (try
    (let [sim      (-> process :sim/_processes only)
          agent    (-> action :agent/_actions solo)
          test     (-> agent :test/_agents solo)
          host     (-> test :test/apiHost)
          device-id (nth-mod (-> agent :agent/deviceIds vec)
                             (-> action :action/index))
          req-id   (-> action :action/clientRequestId)
          email    (if (:agent/identifyEarly agent)
                      (:agent/userEmailAddress agent))
          [time resp] (get-user-token target-host device-id
                                      req-id email)
          user-token  (some-> api-response :reply :user-token)]
      (record action process resp time
              :actionLog/apiResponse (:status resp)
              :actionLog/userToken user-token))))
```

test record

immutable record of all tests

actions, timing

test record

```
(defn record  
  [action process http-response nsec & {:as opts}]  
  ((get sim/*services* :simulant.sim/actionLog)  
   [(gen-log action process http-response nsec opts)]))
```

validations

verify properties

actions and results

global properties

verify properties of prior test executions

validations

Validations are queries.

```

(defn- validate-api-calls
  [db sim val-type attr expected]
  (for [[log actual] (d/q '[:find ?log ?actual
                             :in $ ?sim ?attr ?expected
                             :where
                             [?log :actionLog/sim ?sim]
                             [?log :actionLog/responseMap]
                             [?log ?attr ?actual]]
        db (su/e sim) attr expected)
    :when (not= expected actual)]
    {:action-log-id log
     :validation-type val-type
     :detail (str "Expected " expected ", got " actual)}))

(defn all-api-calls-http-ok [db sim]
  (validate-api-calls db sim
    :all-api-calls-http-ok
    :actionLog/responseCode 200))

```

```
[ :find ?log ?actual
  :in $ ?sim ?attr ?expected
  :where
    [?log :actionLog/sim ?sim]
    [?log :actionLog/responseMap]
    [?log ?attr ?actual]]
db (su/e sim) attr expected
```

test results

record of tests, validations

linked back to execution, model

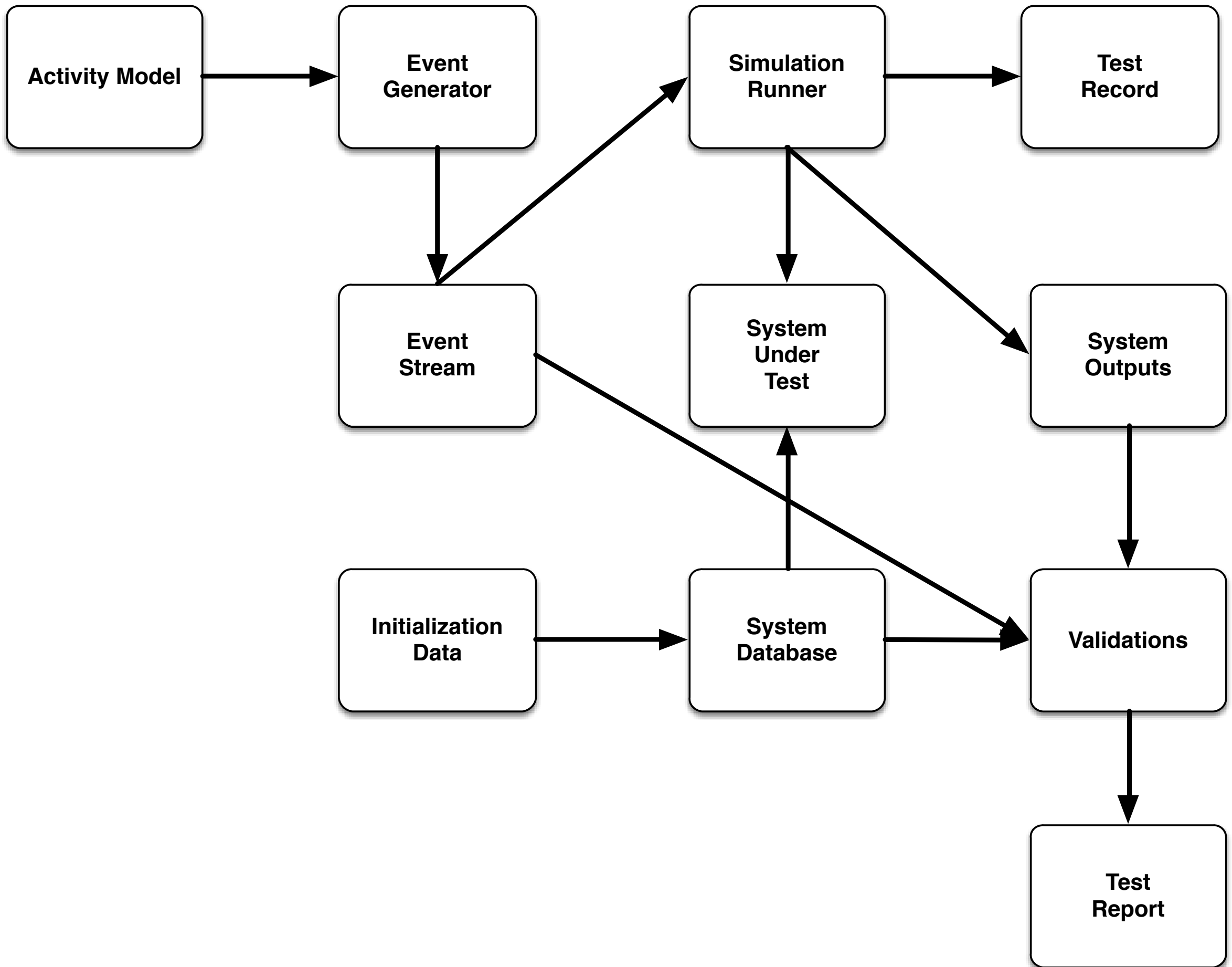
can even record git SHA of model & target codebase

test reports

summary results for consumers

visualization of numeric or statistical information

history, trends, distributions



advantages

coverage!

more exploration than scripted tests

global validation

response time

money in == money out

less expensive than hand-writing tests

considerations

1. what kind of model suffices?
2. containing randomness in generator
3. making actions repeatable
4. target system setup
5. test duration and intensity (and the problem of lag)

Simulant

Some assembly required

<http://github.com/Datomic/simulant>

Simulant provides

Helpers for creating scripts

Execution

Action logging

You provide

A model

Action generators

Validation

-main

A UI...

conclusion

conclusion

simulation testing involves more upfront investment

lifecycle cost is lower

remember: searching for errors, not proving correct

test enough to reach the needed level of confidence

Simulation Testing

@mtnygard



cognitect

Resources

Talk

<https://github.com/clojure/data.generators>. Data generators library.

<https://github.com/clojure/test.generative>. Generative testing library.

<https://github.com/clojure/test.check>. Generative testing library.

<http://clojure.com>. The Clojure language.

<http://www.datomic.com/>. Datomic.

[Finding Race Conditions in Erlang with QuickCheck and PULSE.](#)

Michael Nygard

<https://github.com/mtnygard/presentations/wiki>. Presentations.

<mailto:mtnygard@cognitect.com>