QUASI-FORMAL DESIGN FOR TIME WARP

Brian Beckman

July 9, 2019

Contents

1 INTRODUCTION

Leslie Lamport said, roughly

> *Thinking is good. Writing is Nature's way of showing you how sloppy your thinking is. Mathematics is Nature's way of showing you how sloppy your writing is. Formal mathematics is Nature's way of showing you how sloppy your Mathematics is.*

Formal mathematics is machine-checked mathematics. A *formal spec* is a statement in formal mathematics of your system's static and dynamic properties. In Clojure, specs are Boolean-valued properties—predicate functions—that depend on the internal state variables of the system.

Writing a spec is an art rather than a science. A spec should constrain your system to do what it's supposed to do and to not do what it's not supposed to do. If your spec is too loose, it won't constrain your system. For example, every system trivially satisfies a spec that always says *true*. That's a valid spec, but it's not useful, because your system might crash or go into an infinite loop or launch the missiles, and still satisfy the spec. If your spec is too tight, your system might not generalize well. For example, if you write a spec that requires all outputs from a random-number generator to be positive, then you'll never get a zero or a negative random number. That may be exactly what you want, or it may be a sloppy spec that breaks later when you need non-positive randoms or, worse, doesn't express what you really wanted, which was non-negative randoms.

A *formal verification* or *certification* is a proof that your system satisfies a spec. I define a *quasi-formal* verification as a proof that your system *probably* satisfies a spec. To do a quasi-formal verification, Clojure feeds random data into the system and then checks the spec. Clojure uses your spec to generate random input data. You should formally specify the domains of all your inputs. Domains are, sets, like the integers, or the floating-point numbers, or rows following some SQL schema. Clojure maliciously chooses values from the domains, values likely to cause problems with software in general, like `0`, $-\infty$, `NaN`, empty strings, null pointers, rows with null values in the columns. When Clojure finds values that violate the spec, it *shrinks* them, i.e., searches for nearby but smaller examples that violate the same property. Clojure presents the shrunken cases to you.

Often, quasi-formal verification is the best we can do because a logical proof or an exhaustive test of all possible states of your system is not practically feasible. If either or both are feasible, do them! Yes, really do them! But also do quasi-formal verification because you can do it interactively. At interactive speed, quasi-formal verification is useful because it forces developers to think. An example of exposing subtle bugs in a seemingly trivial program appears in chapter 3.

We have a lot of experience with Time Warp and the aim of this document is to write a great spec for it.

You may skip the warm-ups chapter, 2, unless you want an interactive tutorial about spec.

2 WARM-UPS

Follow along with the URL below. This chapter is mostly code with very little prose because that URL has the prose. I copied the examples here just to limber up my fingers and to get my mind right.

`https://clojure.org/guides/spec`

2.1 Is Clojure working at all?

`C-c C-c` in the following block of code should produce today's date. If clojure is not correctly started, a message will appear in the minibuffer stating `Wrong type argument...`.

```
(java.util.Date.)
```

If you get `Wrong type argument...`, issue emacs command `cider-jack-in`, wait for it to return, then try again. If none of that works, Google about cider and emacs.

That command, if working, will invoke the following project file:

```
(defproject twos-1-10-1 "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "EPL-2.0 OR GPL-2.0-or-later WITH Classpath-exception-2.0"
            :url "https://www.eclipse.org/legal/epl-2.0/"}
  :dependencies [[org.clojure/clojure           "1.10.0"]
                 [org.clojure/test.check        "0.9.0" ]
                 [org.clojure/data.priority-map "0.0.10"]
                 [org.clojure/algo.monads       "0.1.6" ] ]
                 ]
  :main ^:skip-aot twos-1-10-1.core
  :target-path "target/%s"
  :profiles {:uberjar {:aot :all}})
```

(TODO: org-babel-tangle inserts `(ns user)` or `(ns two-1-10-1.core)` at the beginning of the project file, and that may not be suitable. The elisp code that inserts that expression is not easy to find.)

(TODO: this experimental org headline must not have the prefix "COMMENT" lest `org-babel-tangle` not tangle all blocks)

2.2 Require the spec package

```
(require '[clojure.spec.alpha :as s])
```

2.3 Test s/conform

```
(s/conform even? 1000)
```

```
1000
```

```
(s/conform even? 1001)
```

```
:clojure.spec.alpha/invalid
```

2.4 Test s/valid?

```
(s/valid? even? 1000)
```

```
true
```

```
(s/valid? even? 1001)
```

```
false
```

2.5 Test sets as predicates

This import works best when outside the block that follows it

```
(import java.util.Date)
```

```
java.util.Date
```

All the following should be true:

```
(every? true?
        [(s/valid? nil? nil)
         (s/valid? string? "abc")

         (s/valid? #(> % 5) 10)
         (not (s/valid? #(> % 5) 0))

         (s/valid? inst? (java.util.Date.))

         (s/valid? #{:club :diamond :heart :spade} :club)
         (not (s/valid? #{:club :diamond :heart :spade} 42)) ])
```

```
true
```

```
(ns my.domain (:require [clojure.spec.alpha :as s]))
( ->> [ (s/def ::date inst?)
        (s/def ::suit #{:club :diamond :heart :spade})
        (s/valid?  ::date (java.util.Date.))
        (= :club (s/conform ::suit :club)) ]
      (drop 2) (every? true?))
```

```
true
```

2.6 Test doc

(TODO: Sometimes, I cannot access namespace `clojure.repl`. Workaround is to fully qualify the `doc` symbol.)

```
(ns my.domain)
(clojure.repl/doc ::date)
(clojure.repl/doc ::suit)
```

```
-------------------------
:my.domain/date
Spec
  inst?
-------------------------
:my.domain/suit
Spec
  #{:spade :heart :diamond :club}
```

2.7 Test `s/or`

```
(ns my.domain)
( ->> [ (s/def ::name-or-id (s/or :name string? :id int?))

        (s/valid? ::name-or-id "abc")
        (s/valid? ::name-or-id 100)
        (not (s/valid? ::name-or-id :foo)) ]

      (drop 1) (every? true?))
```

```
true
```

2.8 Test `explain`

```
(ns my.domain)
(s/explain ::name-or-id :foo)
```

```
:foo - failed: string? at: [:name] spec: :my.domain/name-or-id
:foo - failed: int? at: [:id] spec: :my.domain/name-or-id
```

```
(ns my.domain)
(clojure.pprint/pprint
  (s/explain-data ::name-or-id :foo))
```

```
#:clojure.spec.alpha{:problems
                     ({:path [:name],
                       :pred clojure.core/string?,
                       :val :foo,
                       :via [:my.domain/name-or-id],
                       :in []}
                      {:path [:id],
                       :pred clojure.core/int?,
                       :val :foo,
                       :via [:my.domain/name-or-id],
                       :in []}),
                     :spec :my.domain/name-or-id,
                     :value :foo}
```

2.9 Test Entity Maps

```
(ns my.domain)
(def email-regex #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
(s/def ::email-type (s/and string? #(re-matches email-regex %)))

(s/def ::acctid     int?)
(s/def ::first-name string?)
(s/def ::last-name string?)
(s/def ::email     ::email-type)

(s/def ::person (s/keys :req [::first-name ::last-name ::email]
                        :opt [::phone]))
(println *ns*)
```

```
#namespace[my.domain]
```

```
(ns my.domain)
(s/valid? ::person
  {::first-name "Bugs"
   ::last-name "Bunny"
   ::email "bugs@example.com"})
```

```
true
```

I can't get the following to word wrap despite `https://www.rosettacode.org/wiki/Word_wrap#Clojure`:

```
(ns my.domain)
(s/explain ::person {::first-name "Bugs"})
```

```
#:my.domain{:first-name "Bugs"} - failed: (contains? % :my.domain/last-name) spec: :my.domain/person
#:my.domain{:first-name "Bugs"} - failed: (contains? % :my.domain/email) spec: :my.domain/person
```

```
(ns my.domain)
(s/explain ::person
           {::first-name "Bugs"
            ::last-name "Bunny"
            ::email "n/a"})
```

```
"n/a" - failed: (re-matches email-regex %) in: [:my.domain/email] at: [:my.domain/email] spec: :my.domain/e
```

```
(ns my.domain)
(s/def :unq/person
  (s/keys :req-un [::first-name ::last-name ::email]
          :opt-un [::phone]))

(s/conform :unq/person
           {:first-name "Bugs"
            :last-name "Bunny"
            :email "bugs@example.com"})
;;=> {:first-name "Bugs", :last-name "Bunny", :email "bugs@example.com"}
```

```
:unq/person{:first-name "Bugs", :last-name "Bunny", :email "bugs@example.com"}
```

```
(ns my.domain)
(s/explain :unq/person
           {:first-name "Bugs"
            :last-name "Bunny"
            :email "n/a"})
;; "n/a" - failed: (re-matches email-regex %) in: [:email] at: [:email]
;;   spec: :my.domain/email-type

(s/explain :unq/person
           {:first-name "Bugs"})
;; {:first-name "Bugs"} - failed: (contains? % :last-name) spec: :unq/person
;; {:first-name "Bugs"} - failed: (contains? % :email) spec: :unq/person
```

```
"n/a" - failed: (re-matches email-regex %) in: [:email] at: [:email] spec: :my.domain/email-type
{:first-name "Bugs"} - failed: (contains? % :last-name) spec: :unq/person
{:first-name "Bugs"} - failed: (contains? % :email) spec: :unq/person
```

If the preceding two are run without `(ns my.domain)`, the last one reports `Success!`. Why? Because the spec, if evaluated in the default namespace `twos-1-10-1.core` merely demands the presence of the unqualified keyword `:email`, "unqualified" meaning "not in the namespace." Because there is no conformance spec `::email` in `twos-1-10-1.core`, Clojure.spec doesn't do a deeper check.

We disable the evaluation of these blocks because evaluating them messes up the internal state of Clojure.spec and requires us to re-evaluate things above. Just remember that namespaces are tricky; the authors of Clojure admit so: `https://clojure.org/guides/repl/navigating_namespaces`.

NOTICE `:eval never` and `begin_example` for the following. Do not evaluate them.

```
(s/def :unq/person
  (s/keys :req-un [::first-name ::last-name ::email]
          :opt-un [::phone]))

(s/conform :unq/person
           {:first-name "Bugs"
            :last-name "Bunny"
            :email "bugs@example.com"})
;;=> {:first-name "Bugs", :last-name "Bunny", :email "bugs@example.com"}
```

```
: :unq/person{:first-name "Bugs", :last-name "Bunny", :email "bugs@example.com"}
```

```
(s/explain :unq/person
           {:first-name "Bugs"
            :last-name "Bunny"
            :email "n/a"})
;; "n/a" - failed: (re-matches email-regex %) in: [:email] at: [:email]
;;   spec: :my.domain/email-type

(s/explain :unq/person
           {:first-name "Bugs"})
;; {:first-name "Bugs"} - failed: (contains? % :last-name) spec: :unq/person
;; {:first-name "Bugs"} - failed: (contains? % :email) spec: :unq/person
```

```
: Success!
: {:first-name "Bugs"} - failed: (contains? % :last-name) spec: :unq/person
: {:first-name "Bugs"} - failed: (contains? % :email) spec: :unq/person
```

2.10 Test records

```
(ns my.domain)
(def email-regex #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
(s/def ::email-type (s/and string? #(re-matches email-regex %)))

(s/def ::acctid     int?)
(s/def ::first-name string?)
(s/def ::last-name string?)
(s/def ::email     ::email-type)

(s/def ::person (s/keys :req [::first-name ::last-name ::email]
                        :opt [::phone]))
(println *ns*)
```

```
#namespace[my.domain]
```

```
  (ns my.domain)
  (defrecord Person [first-name last-name email phone])

  (s/explain :unq/person
             (->Person "Bugs" nil nil nil))
;; nil - failed: string? in: [:last-name] at: [:last-name] spec: :my.domain/last-name
;; nil - failed: string? in: [:email] at: [:email] spec: :my.domain/email-type
```

```
nil - failed: string? in: [:last-name] at: [:last-name] spec: :my.domain/last-name
nil - failed: string? in: [:email] at: [:email] spec: :my.domain/email-type
```

```
(ns my.domain)
(s/conform :unq/person
  (->Person "Bugs" "Bunny" "bugs@example.com" nil))
```

```
#my.domain.Person{:first-name "Bugs", :last-name "Bunny", :email "bugs@example.com", :phone nil}
```

2.11 Test keyword args

```
(ns my.domain)
(s/def ::port number?)
(s/def ::host string?)
(s/def ::id keyword?)
(s/def ::server (s/keys* :req [::id ::host] :opt [::port]))
(clojure.pprint/pprint
 (s/conform ::server [::id :s1 ::host "example.com" ::port 5555]))
```

```
#:my.domain{:id :s1, :host "example.com", :port 5555}
```

2.12 Test key-spec merges

```
(ns my.domain)
(s/def :animal/kind string?)
(s/def :animal/says string?)
(s/def :animal/common (s/keys :req [:animal/kind :animal/says]))
(s/def :dog/tail? boolean?)
(s/def :dog/breed string?)
(s/def :animal/dog (s/merge :animal/common
                            (s/keys :req [:dog/tail? :dog/breed])))
(println (s/valid? :animal/dog
                   {:animal/kind "dog"
                    :animal/says "woof"
                    :dog/tail? true
                    :dog/breed "retriever"}))
```

```
true
```

Notice the specs above are not in the namespace.

```
; (ns my.domain) ;; <-- UNCOMMENT to make an error
(clojure.repl/doc :animal/kind)
```

```
-------------------------
:animal/kind
Spec
  string?
```

2.13 Test multi-spec

```
(ns my.domain)
(s/def :event/type keyword?)
(s/def :event/timestamp int?)
(s/def :search/url string?)
(s/def :error/message string?)
(s/def :error/code int?)

(defmulti event-type :event/type)
(defmethod event-type :event/search [_]
  (s/keys :req [:event/type :event/timestamp :search/url]))
(defmethod event-type :event/error [_]
  (s/keys :req [:event/type :event/timestamp :error/message :error/code]))

(s/def :event/event (s/multi-spec event-type :event/type))

(println
 (every? true?
         [(s/valid? :event/event
                    {:event/type :event/search
                     :event/timestamp 1463970123000
                     :search/url "https://clojure.org"})

          (s/valid? :event/event
                    {:event/type :event/error
                     :event/timestamp 1463970123000
                     :error/message "Invalid host"
                     :error/code 500})]))
```

```
true
```

```
(ns my.domain)
(s/explain :event/event
  {:event/type :event/restart})
```

```
Success!
```

```
(ns my.domain)
(s/explain :event/event
  {:event/type :event/search
   :search/url 200})
```

```
200 - failed: string? in: [:search/url] at: [:event/search :search/url] spec: :search/url
{:event/type :event/search, :search/url 200} - failed: (contains? % :event/timestamp) at: [:event/search] s
```

2.13.1 Open types

Add a new type to `:event/event` above:

```
(ns my.domain)
(defmethod event-type :event/restart [_]
  (s/keys :req [:event/type]))
(println (s/valid? :event/event
                   {:event/type :event/restart}))
```

```
true
```

2.14 Test collections

2.14.1 homogeneous small: coll-of

```
(ns my.domain)
[ (s/conform (s/coll-of keyword?) [:a :b :c])
  (s/conform (s/coll-of number?) #{ 5 10  2}) ]
```

```
'((:a :b :c) #(2 5 10))
```

```
(ns my.domain)
(s/def ::vnum3 (s/coll-of number? :kind vector?
                                  :count 3
                                  :distinct true
                                  :min-count 3 ;; redundant but harmless ...
                                  :max-count 3 ;; ... here as a reminder
                                  :into #{}))
(s/conform ::vnum3 [ 5 10  2])
```

```
:my.domain/vnum3#{2 5 10}
```

Notice that, in the last failing example, only the `distinc?` spec is reported:

```
(ns my.domain)
(s/explain ::vnum3 #{5 10 2})
(s/explain ::vnum3 [1 1 2])
(s/explain ::vnum3 [1 2 :a])
(s/explain ::vnum3 [1])
(s/explain ::vnum3 [1 1 :a])
```

```
#{2 5 10} - failed: vector? spec: :my.domain/vnum3
[1 1 2] - failed: distinct? spec: :my.domain/vnum3
:a - failed: number? in: [2] spec: :my.domain/vnum3
[1] - failed: (= 3 (count %)) spec: :my.domain/vnum3
[1 1 :a] - failed: distinct? spec: :my.domain/vnum3
```

2.14.2 homogeneous large: every, every-kv

```
s/*coll-check-limit*
```

```
101
```

(TODO: I expected the following to return a set and therefore not to require the exterior call of `distinct`.)

(TODO: I expected the following to sample `s/*coll-check-limit*`, that is, 101, by default, elements of the infinite collection `(repeat 42)`, and thus, to terminate. It (apparently) doesn't terminate if the `(take 1000 ...)` wrapper is removed.)

```
(ns my.domain)
(distinct (s/conform
            (s/every int? :kind vector :into #{})
            (take 1000 (repeat 42))))
```

```
42
```

2.14.3 heterogeneous: tuple

```
(ns my.domain)
(s/def ::point (s/tuple double? int? double? keyword?))
(s/conform ::point [1.5 42 -0.5 :ok])
```

```
:my.domain/point[1.5 42 -0.5 :ok]
```

```
(s/conform (s/cat :x double? :h int? :y double? :kw keyword?) [1.5 42 -0.5 :ok])
```

```
:x   1.5   :h   42   :y   -0.5   :kw   :ok
```

2.14.4 homogenous: map-of

```
(ns my.domain)
(s/def ::scores (s/map-of string? int?))
(s/conform ::scores {"Sally" 1000, "Joe" 500})
```

```
:my.domain/scores{"Sally" 1000, "Joe" 500}
```

> By default map-of will validate but not conform keys because conformed keys might create key duplicates that would cause entries in the map to be overridden. If conformed keys are desired, pass the option :conform-keys true.

2.15 Test sequences

```
(ns my.domain)
(s/def ::ingredient (s/cat :quantity number? :unit keyword?))
(s/conform ::ingredient [2 :teaspoon])
```

```
:my.domain/ingredient{:quantity 2, :unit :teaspoon}
```

```
(ns my.domain)
(s/explain ::ingredient [11 "peaches"])
```

```
"peaches" - failed: keyword? in: [1] at: [:unit] spec: :my.domain/ingredient
```

```
(ns my.domain)
(s/explain ::ingredient ["peaches"])
```

```
"peaches" - failed: number? in: [0] at: [:quantity] spec: :my.domain/ingredient
```

2.16 Test nested regexes (regices?)

```
(ns my.domain)
(s/def ::nested
  (s/cat :names-kw #{:names}
         :names    (s/spec (s/* string?))
         :nums-kw  #{:nums}
         :nums     (s/spec (s/* number?))))
(s/conform ::nested [:names ["a" "b"], :nums [1 2 3]])
```

```
:my.domain/nested{:names-kw :names, :names ["a" "b"], :nums-kw :nums, :nums [1 2 3]}
```

2.17 Test runtime validation (:pre and :post)

Without the `println`, the following produces a namespaced object containing a string.

```
(ns my.domain)
(defn person-name
  [person]
  {:pre [(s/valid? ::person person)]
   :post [(s/valid? string? %)]}
  (str (::first-name person) " " (::last-name person)))
(println (person-name {::first-name "Bugs"
                       ::last-name "Bunny"
                       ::email "bugs@example.com"}))
```

```
Bugs Bunny
```

```
(ns my.domain)
(defn person-name
  [person]
  (let [p (s/assert ::person person)]
    (str (::first-name p) " " (::last-name p))))

(s/check-asserts true) ;; <~~ Don’t forget this; it’s off by default.
(person-name 100)
```

```
class clojure.lang.ExceptionInfoclass clojure.lang.ExceptionInfoExecution error - invalid arguments to my.d
100 - failed: map?
```

```
(ns my.domain)
(s/def ::config (s/*
                 (s/cat :prop string?
                        :val  (s/alt :s string? :b boolean?))))
(clojure.pprint/pprint
  (s/conform ::config ["-server" "foo" "-verbose" true "-user" "joe"]))
```

```
[{:prop "-server", :val [:s "foo"]}
 {:prop "-verbose", :val [:b true]}
 {:prop "-user", :val [:s "joe"]}]
```

```
(ns my.domain)
(defn- set-config [prop val]
  ;; dummy fn
  (println "set" prop val))

(defn configure [input]
  (let [parsed (s/conform ::config input)]
    (if (= parsed ::s/invalid)
      (throw (ex-info "Invalid input" (s/explain-data ::config input)))
      (for [{prop :prop [_ val] :val} parsed]
        (set-config (subs prop 1)  ;; Strip the leading hyphen
                    val)))))

(configure ["-server" "foo" "-verbose" true "-user" "joe"])
```

```
set server foo
set verbose true
set user joe
```

2.18 Test fdef [sic; not =ifdef=]

```
(ns my.domain)
(defn ranged-rand
  "Returns random int in range start <= rand < end. Noti"
  [start end]
  (+ start (long (rand (- end start)))))

```

```
(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               #(< (:start %) (:end %)))
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))
```

```
#’my.domain/ranged-randmy.domain/ranged-rand
```

```
(ns my.domain)
(clojure.repl/doc my.domain/ranged-rand)
```

```
-------------------------
my.domain/ranged-rand
([start end])
  Returns random int in range start <= rand < end. Noti
Spec
  args: (and (cat :start int? :end int?) (< (:start %) (:end %)))
  ret: int?
  fn: (and (>= (:ret %) (-> % :args :start)) (< (:ret %) (-> % :args :end)))
```

```
(ns my.domain)
(defn adder [x] #(+ x %))

(s/fdef adder
  :args (s/cat :x number?)
  :ret (s/fspec :args (s/cat :y number?)
                :ret number?)
  :fn #(= (-> % :args :x) ((:ret %) 0)))

(clojure.repl/doc my.domain/adder)
```

```
-------------------------
my.domain/adder
([x])
Spec
  args: (cat :x number?)
  ret: (fspec :args (cat :y number?) :ret number? :fn nil)
  fn: (= (-> % :args :x) ((:ret %) 0))
```

2.19 Card game

```
  (ns my.domain)
  (def suit? #{:club :diamond :heart :spade})
  (def rank? (into #{:jack :queen :king :ace} (range 2 11)))
  (def deck (for [suit suit? rank rank?] [rank suit]))

  (s/def ::card (s/tuple rank? suit?))
  (s/def ::hand (s/* ::card))

  (s/def ::name string?)
  (s/def ::score int?)
  (s/def ::player (s/keys :req [::name ::score ::hand]))

  (s/def ::players (s/* ::player))
  (s/def ::deck (s/* ::card))
  (s/def ::game (s/keys :req [::players ::deck]))

  (def kenny
    {::name "Kenny Rogers"
     ::score 100
     ::hand []})
  (println (s/valid? ::player kenny))

(s/explain ::game
  {::deck deck
   ::players [{::name "Kenny Rogers"
               ::score 100
               ::hand [[2 :banana]]}]})
```

```
true
:banana - failed: suit? in: [:my.domain/players 0 :my.domain/hand 0 1] at: [:my.domain/players :my.domain/h
```

2.20 Testing test.check

2.20.1 Basic generators

```
(ns my.domain)
(require '[clojure.spec.gen.alpha :as gen])
(clojure.pprint/pprint
  [ (gen/generate (s/gen int?))
    (gen/generate (s/gen nil?))
    (gen/sample   (s/gen string?))
    (gen/sample   (s/gen (s/cat :k keyword? :nums (s/* number?))) 5)
    (s/exercise   (s/cat :k keyword? :ns (s/* number?)) 5)
    (gen/sample   (s/gen (s/and int? #(> % 0) #(zero? (mod % 3)))))
    ; (gen/generate (s/gen ::player)) ;; <o=-< works, but is too long
    ; (gen/generate (s/gen ::game)) ;; <o=-<
  ])
```

```
[-2010
 nil
 ("" "" "" "" "K" "dV" "xie" "5utRr8" "DU3" "0rB")
 ((:L)
  (:iX)
  (:uWk/!9b)
  (:Z?+0/f_Z)
  (:!?*D!.C5.lc.Y/saf9 -0.5 -1 1.5 3.125))
 ([(:H) {:k :H}]
  [(:W! -0.5) {:k :W!, :ns [-0.5]}]
  [(:_e?/q -1) {:k :_e?/q, :ns [-1]}]
  [(:c/MM -1 ##-Inf -0.5) {:k :c/MM, :ns [-1 ##-Inf -0.5]}]
  [(:gz1.?+._B!.G?*B/aB6+ -2 1)
   {:k :gz1.?+._B!.G?*B/aB6+, :ns [-2 1]}])
 (9 12 33 3 60 57 15 20841 405 45)]
```

With fully qualified symbols everywhere:

```
(clojure.repl/doc my.domain/ranged-rand)
```

```
-------------------------
my.domain/ranged-rand
([start end])
  Returns random int in range start <= rand < end. Noti
Spec
  args: (and (cat :start int? :end int?) (< (:start %) (:end %)))
  ret: int?
  fn: (and (>= (:ret %) (-> % :args :start)) (< (:ret %) (-> % :args :end)))
```

```
(ns my.domain)
(clojure.pprint/pprint
  (s/exercise-fn ‘ranged-rand)) ;; TODO: <o=-< quote doesn’t work; only
                                ;; backtick, which isn’t =quasiquote= here
```

```
([(6 7) 6]
 [(-1 0) -1]
 [(-1 1) -1]
 [(-8 -4) -6]
 [(-3 -1) -2]
 [(-1 0) -1]
 [(-7 30) 16]
 [(-32 2) -12]
 [(-204 -99) -142]
 [(-1 12) 3])
```

2.20.2 Testing `s/with-gen`

Keyword generator search space is too large; with overwhelming probability (monkeys on keyboards and Jose Luis Borges notwithstanding), we’re not going to generate keywords in our namespace:

```
(ns my.domain)
(s/def ::kws (s/and
              keyword?
              #(= (namespace %) "my.domain")))
(s/valid? ::kws :my.domain/name) ;; true
(gen/sample (s/gen ::kws)) ;; overwhelmingly unlikely we’ll generate useful
                           ;; keywords this way
```

```
:my.domain/kwstrueclass clojure.lang.ExceptionInfoclass clojure.lang.ExceptionInfoError printing return val
Couldn’t satisfy such-that predicate after 100 tries.
```

To generate useful samples, reduce the size of the keyword gen space by supplying an explicit set of keywords, all of which are in the namespace. The set is, itself, a predicate, thus a correct argument for `s/gen`. Define `kw-gen` to be that hand-written set of keywords.

```
(ns my.domain)
(def kw-gen (s/gen #{:my.domain/->Person :my.domain/rank? :my.domain/person-name
                     :my.domain/email-regex :my.domain/deck :my.domain/configure
                     :my.domain/-syms :my.domain/map->Person :my.domain/adder
                     :my.domain/kenny :my.domain/ranged-rand :my.domain/event-type
                     :my.domain/kw-gen :my.domain/suit?}))
(clojure.pprint/pprint
  (gen/sample kw-gen 5))
```

```
(:my.domain/deck
 :my.domain/person-name
 :my.domain/event-type
 :my.domain/map->Person
 :my.domain/rank?)
```

Now try `with-gen`, specifying the keyword gen-space by hand, not using `kw-gen`, defined one block above. The final argument to `s/with-gen` must be a thunk (function of no arguments) wrapping the generator:

```
(ns my.domain)
(s/def ::kws (s/with-gen
               (s/and keyword? #(= (namespace %) "my.domain"))
               #(s/gen #{:my.domain/->Person :my.domain/rank? :my.domain/person-name
                         :my.domain/email-regex :my.domain/deck :my.domain/configure
                         :my.domain/-syms :my.domain/map->Person :my.domain/adder
                         :my.domain/kenny :my.domain/ranged-rand :my.domain/event-type
                         :my.domain/kw-gen :my.domain/suit?}
                       )))
(clojure.pprint/pprint
  (gen/sample (s/gen ::kws) 5))
```

```
(:my.domain/rank?
 :my.domain/person-name
 :my.domain/-syms
 :my.domain/rank?
 :my.domain/suit?)
```

Now try `with-gen`, specifying the keyword gen-space by wrapping the reference to `kw-gen`, defined two blocks above, in a thunk:

```
(ns my.domain)
(s/def ::kws (s/with-gen
               (s/and keyword? #(= (namespace %) "my.domain"))
               (fn [] kw-gen)))
(clojure.pprint/pprint
  (gen/sample (s/gen ::kws) 5))
```

```
(:my.domain/->Person
 :my.domain/kenny
 :my.domain/person-name
 :my.domain/suit?
 :my.domain/deck)
```

Generalize by sucking all symbols out of the actual namespace, not writing them out by hand:

```
(ns my.domain)
(def -kwds (into #{} (map #(keyword "my.domain" (str %))
                          (keys (ns-publics 'my.domain)))))
(def kw-gen-2 (s/gen -kwds))
(s/def
  ::kws
  (s/with-gen
    (s/and keyword? #(= (namespace %) "my.domain"))
    (fn [] kw-gen-2)))
(clojure.pprint/pprint (gen/sample (s/gen ::kws) 5))
```

```
(:my.domain/kenny
 :my.domain/deck
 :my.domain/configure
 :my.domain/email-regex
 :my.domain/-kwds)
```

2.20.3 Open generator spaces with fmap

```
(ns my.domain)
(let [digit? (set (range 0 10))
      ascint #(- (int %) 48)]
  (clojure.pprint/pprint
           ( ->>
            (gen/string-alphanumeric)
            (gen/such-that
             #(and (not= % "")
                   (not (digit? (ascint (first %))))))
            (gen/fmap #(keyword "my.domain" %))
            gen/sample)))
```

```
(:my.domain/0
 :my.domain/C
 :my.domain/H8
 :my.domain/On2
 :my.domain/AaYy
 :my.domain/FFCCQA
 :my.domain/Zn8s
 :my.domain/zw
 :my.domain/h75VT35m
 :my.domain/w3Jxw)
```

```
(ns my.domain)
(s/def ::hello
  (s/with-gen
   #(clojure.string/includes? % "hello")
   #(gen/fmap (fn [[s1 s2]] (str s1 "hello" s2))
            (gen/tuple (gen/string-alphanumeric)
                    (gen/string-alphanumeric)))))
(clojure.pprint/pprint
 (gen/sample (s/gen ::hello)))
```

```
("hello"
 "hello"
 "hellooD"
 "L54hello4b"
 "hellosRwx"
 "39VGshello"
 "Xn0hello79a00"
 "I0471oChellom"
 "Qbhello"
 "50hellonEer01")
```

2.20.4 Range specs and generators

```
(ns my.domain)
(-> (s/int-in 0 11)
   s/gen
   gen/sample)
```

0 1 2 1 3 1 4 4 1 3

```
(ns my.domain)
(-> (s/inst-in #inst "2000" #inst "2010")
   s/gen
   (gen/sample 55)
   ((partial take-last 5))
   clojure.pprint/pprint
)

```

```
(#inst "2000-01-01T00:03:11.474-00:00"
 #inst "2000-01-02T08:13:34.406-00:00"
 #inst "2000-01-01T00:00:00.133-00:00"
 #inst "2000-01-01T00:00:00.795-00:00"
 #inst "2000-05-12T07:04:26.138-00:00")
```

2.21 Instrumentation and Testing

`Ranged-rand` is an interesting function. It's defined as follows

$$\mathrm{rr}(s, e) = s + \mathrm{rand}(e - s) \tag{1}$$

where

$$\text{rand}(n) = n * \mathtt{rand}([0..1)) \tag{2}$$

and $\mathtt{rand}([0..1))$ means *a random number between 0, inclusive, and 1, exclusive.*

The intent is obvious when $s < e$ and both are not negative, implying that $e - s > 0$. It's what we normally mean by a *range from* s *to* e. With Clojure we can spec that intent: remember

```
(ns my.domain)
(defn ranged-rand
  "Returns random int in range start <= rand < end. Noti"
  [start end]
  (+ start (long (rand (- end start)))))

(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               #(not (neg? (:start %))) #(not (neg? (:end %)))
               #(< (:start %) (:end %)))
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))
```

By instrumenting the function, we can check its spec at run time. This is expensive, so not a default:

```
(ns my.domain)
(require '[clojure.spec.test.alpha :as stest])
(stest/instrument `ranged-rand)
(-> (ranged-rand 8 5)
    clojure.pprint/pprint)
(-> (ranged-rand -42 0)
    clojure.pprint/pprint)
```

```
class clojure.lang.ExceptionInfoclass clojure.lang.ExceptionInfoclass clojure.lang.ExceptionInfoclass cloju
{:start 8, :end 5} - failed: (< (:start %) (:end %))
Execution error - invalid arguments to my.domain/ranged-rand at (form-init12747529328508439197.clj:7).
{:start -42, :end 0} - failed: (not (neg? (:start %)))
```

If we `unstrument` the function, we can get away with weird arguments:

```
(ns my.domain)
(stest/unstrument `ranged-rand)
(-> (ranged-rand 8 5)
    clojure.pprint/pprint)
(-> (ranged-rand -42 0)
    clojure.pprint/pprint)
```

```
7
-11
```

Should we spec the behavior when `start` is greater than or equal to `end` and when either or both are negative?

We defined `ranged-rand`, mathematically, as $s + d \times [0..1)$, where $d = e - s$ and $[0..1)$ stands for a uniform sample between 0, inclusive, and 1, exclusive (it takes digging into the source for `clojure.core/rand` to bottom-out this definition in `java.lang.Math/random`):

```
;; from clojure.core
(defn rand
  "Returns a random floating point number between 0 (inclusive) and
  n (default 1) (exclusive)."
  {:added "1.0"
   :static true}
  ([] (. Math (random)))
  ([n] (* n (rand))))
```

This definition is meaningful and even seems reasonable for s, d, d negative or 0. Let's do a relaxed spec, which only checks `int?` types for arguments and the `:fn` invariant on `:ret`, and generate some values:

```
(ns my.domain)
(defn ranged-rand
  "Returns random int in range start <= rand < end. Noti"
  [start end]
  (+ start (long (rand (- end start)))))

(s/fdef ranged-rand
  :args (s/cat :start int? :end int?)
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))

(-> ‘ranged-rand
    s/exercise-fn
    clojure.pprint/pprint)
```

```
([(0 -1) 0]
 [(-1 0) -1]
 [(-1 -1) -1]
 [(-2 -2) -2]
 [(2 3) 2]
 [(-2 1) 0]
 [(-24 -1) -7]
 [(-59 -16) -35]
 [(-15 13) -7]
 [(0 0) 0])
```

3 TESTING

Testing is the big payoff for `spec`. Probabilistic esting is the best we can do without a formal proof or an exhaustive test.

It is perhaps surprising and certainly instructive that `ranged-rand` has bugs, and that writing and checking a good spec reveals the bugs, and that fixing the spec controls the bugs.

3.1 Original spec reveals a bug

Here is the original code for `ranged-rand`. You might think this is so trivial that it doesn’t need a spec. But there are bugs. Can you spot them before you go on?

```
(defn ranged-rand
  "Return a random int in range start <= rand < end."
  [start end]
  (+ start (long (rand (- end start)))))
```

Let’s check the original spec, from the official Clojure docs, which didn’t have a constraint for `start` and `end` other than they be `ints`. Lengthen the test to 100, 000 trials so that we’re almost certain to trip the unforeseen bug:

```
(ns my.domain)

(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               ;; DON’T CONSTRAIN #(not (neg? (:start %))) #(not (neg? (:end %)))
               #(< (:start %) (:end %)))
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))

(-> (stest/check ‘ranged-rand
                 {:clojure.spec.test.check/opts
                  {:num-tests 100000}})
    first
    stest/abbrev-result
    :failure .getMessage ;; <o=-< That’s a java.lang.Throwable method
                         ;; <o=-< Remove that line to see everything!
    clojure.pprint/pprint)
```

```
"integer overflow"
```

The complete output is very long and includes a stack trace, which clutters up the document, so I filter the output with `:failure` and `.getMessage`. We can see that (AHA!) `start` and `end` can be so far apart that their difference is too big for a `clojure.core$long`. Quoting the document for `spec` `https://clojure.org/guides/spec`:

> *A keen observer will notice that `ranged-rand` contains a subtle bug. If the difference between `start` and `end` is very large (larger than is representable by `Long/MAX_VALUE`), then `ranged-rand` will produce an `IntegerOverflowException`. If you run `check` several times you will eventually cause this case to occur.*

3.2 Constrained spec fixes the bug

Our more constrained spec doesn’t fail that check. The following takes a long time to run, and really only runs in the REPL, not in org-babel, so we just paste the results of one run in this document in an `example` block:

```
(ns my.domain)

```

```
(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               ; OH YES, HERE IS THE FIX, NOT TO THE CODE, BUT TO THE SPEC
               #(not (neg? (:start %))) #(not (neg? (:end %)))
               #(< (:start %) (:end %)))
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))

(-> (stest/check ‘ranged-rand
                 {:clojure.spec.test.check/opts
                  {:num-tests 100000}})
    clojure.pprint/pprint)
```

```
({:spec
  #object[clojure.spec.alpha$fspec_impl$reify__2524 0x9206636 "clojure.spec.alpha$fspec_impl$reify__2524@92
  :clojure.spec.test.check/ret
  {:result true, :num-tests 100000, :seed 1562631597111},
  :sym my.domain/ranged-rand})
```

3.3 Relaxed spec has a different bug

Consider a relaxed spec, which doesn't check that start < end, but fails the check:

```
(ns my.domain)

(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               #(not (neg? (:start %))) #(not (neg? (:end %))))
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))

(-> (stest/check ‘ranged-rand
                 {:clojure.spec.test.check/opts
                  {:num-tests 1001}})
    first
    stest/abbrev-result
    :failure ::s/problems ;; <o=-< a new filter!
    clojure.pprint/pprint)
```

```
[{:path [:fn],
  :pred
  (clojure.core/fn
   [%]
   (clojure.core/< (:ret %) (clojure.core/-> % :args :end))),
  :val {:args {:start 0, :end 0}, :ret 0},
  :via [],
  :in []}]
```

We see that, although the return value is sensible when `start` equals `end`, it's out of spec and not very useful. Put in the constraint that `start` not equal `end`, but still allow `start` to be greater than `end`. That's both sensible and useful, if a little “creative.” The proper inclusion test becomes more delicate, however. In the normal case, where start is less than end, we’re closed on start and open on end, as before. In the reversed case, however, we’re closed on the right, at start, and open on the left, at end.

```
(ns my.domain)

(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               #(not (neg? (:start %))) #(not (neg? (:end %)))
               #(not= (:start %) (:end %)))
  :ret int?

  :fn (s/or :regular-branch
            (s/and
             #(< (-> % :args :start) (-> % :args :end))
             #(>= (:ret %) (-> % :args :start))
             #(<  (:ret %) (-> % :args :end)))
            :reversed-branch
            (s/and
             #(> (-> % :args :start) (-> % :args :end))
             #(<= (:ret %) (-> % :args :start))
             #(>  (:ret %) (-> % :args :end)))
            ))

(-> (stest/check ‘ranged-rand
                 {:clojure.spec.test.check/opts
                  {:num-tests 1001}})
    first
    clojure.pprint/pprint)
```

```
{:spec
 #object[clojure.spec.alpha$fspec_impl$reify__2524 0x73ee284 "clojure.spec.alpha$fspec_impl$reify__2524@73e
 :clojure.spec.test.check/ret
 {:result true, :num-tests 1001, :seed 1562720011652},
 :sym my.domain/ranged-rand}
```

All of this isn’t worth the effort for this specific, practical case. But it’s a useful exercise to show two things:

1. Formally spec’cing even seemingly easy code is surprisingly difficult and forces you to *think* below the surface. Without this thinking, we would have put the original code into production with at least two bugs because we *thought*, superficially, we knew what we were doing. The exercise of spec’cing forced us to question our smug assuredness.

2. Checking your specs reveals how sloppy even your deeper thinking is. The more delicate inclusion testing took a couple of rounds to get right, and it wouldn’t have been right without check’s quasi-verification to reveal problems.

Clojure.spec only gives us quasi-formal checking: we don’t have a theorem, though I think it wouldn’t be too hard to drive to one at this point. But the checks are extremely useful, much more useful than mere unit testing, because they force us to consider and encode subtleties. The goal is to cover *all* subtleties, and quasi-verification gives us a better chance of getting there.

4 TODO: ORCHESTRA (BEYOND INSTRUMENT) AND EXPOUND (BEYOND EXPLAIN)

5 TODO: ENUMERATE NAMESPACE

5.1 CORE

```
(ns my.domain)
(-> (stest/enumerate-namespace ’clojure.core)
    stest/check
    clojure.pprint/pprint )
```

```
()
```

5.2 SPEC.ALPHA

```
(ns my.domain)
(-> (stest/enumerate-namespace ’my.domain)
    ;stest/check
    clojure.pprint/pprint )
```

```
#{my.domain/event-type my.domain/person-name my.domain/kenny
  my.domain/rank? my.domain/-kwds my.domain/ranged-rand
  my.domain/email-regex my.domain/deck my.domain/suit?
  my.domain/configure my.domain/adder my.domain/map->Person
  my.domain/set-config my.domain/->Person my.domain/kw-gen-2
  my.domain/kw-gen}
```

5.3 MONADS

```
(ns my.domain)
(require ’[clojure.algo.monads :as m])
(-> (stest/enumerate-namespace ’clojure.algo.monads)
    stest/check
    clojure.pprint/pprint )
```

```
()
```

6 ARDES URLS

ARDES 101

https://w.amazon.com/bin/view/Amazon_Robotics/Virtual_Systems/Get_Started

ARDES 2.0 SDK

https://w.amazon.com/bin/view/Amazon_Robotics/Virtual_Systems/Engines/ARDES/SDK2.0/

ARDES AirGateway Simulation

https://drive.corp.amazon.com/documents/OpsSimulation/AR%20ARDES%20AirGateway%20Simulation.docx

ARDES Batch Interface

https://w.amazon.com/index.php/Amazon%20Robotics/Virtual%20Systems/Developers/ArdesBatch

ARDES CLI Command Reference

https://w.amazon.com/index.php/Main/ARDES/Internal/ArdesCLICommandReference

ARDES Case Depalletizer Simulation

https://drive.corp.amazon.com/documents/OpsSimulation/AR%20ARDES%20Case%20Depalletizer%20Simulation.docx

ARDES Developer Onboarding

https://w.amazon.com/bin/view/Main/ARDES/Dev/Onboarding/#HRunyourfirstlocalsimulation

ARDES FC Rolo Simulation

https://drive.corp.amazon.com/documents/OpsSimulation/AR%20ARDES%20FC%20Rolo%20Simulation.docx

ARDES Internal Visualization

https://w.amazon.com/bin/view/Main/ARDES/Internal#HVisualization

ARDES Parallel Event Coordinator

https://w.amazon.com/index.php/Amazon%20Robotics/Virtual%20Systems/Developers/ParallelEventProcessing

ARDES Quick Start for Mac

https://w.amazon.com/bin/view/Main/ARDES/demo/

ARDES ROLO (Restowing of Relocated Inventory)

https://w.amazon.com/bin/view/Amazon_Robotics/Virtual_Systems/Engines/ARDES/ROLO/

ARDES SortCenter Simulation

https://drive.corp.amazon.com/documents/OpsSimulation/AR%20ARDES%20SortCenter%20Simulation.docx

ARDES Streaming Service

https://code.amazon.com/packages/ARDESStreamingServiceService/blobs/mainline/--/install_ARDESStreamingServi
workspace.sh?raw=1

ARDES Time Warp

https://w.amazon.com/bin/view/Amazon_Robotics/Virtual_Systems/Engines/Interns/TimeWarp/

Black Caiman

https://w.amazon.com/bin/view/Black_Caiman/

Study of FlexSim / ARDES integration

https://w.amazon.com/index.php/Amazon%20Robotics/Virtual%20Systems/Developers/ArdesFlexSimIntegration