

# QUASI-FORMAL DESIGN FOR TIME WARP

Brian Beckman

July 9, 2019

## Contents

<b>1</b>	<b>INTRODUCTION</b>	<b>2</b>
<b>2</b>	<b>WARM-UPS</b>	<b>3</b>
2.1	Is Clojure working at all? . . . . .	3
2.2	Require the spec package . . . . .	4
2.3	Test s/conform . . . . .	4
2.4	Test s/valid? . . . . .	4
2.5	Test sets as predicates . . . . .	4
2.6	Test doc . . . . .	5
2.7	Test s/or . . . . .	5
2.8	Test explain . . . . .	6
2.9	Test Entity Maps . . . . .	6
2.10	Test records . . . . .	8
2.11	Test keyword args . . . . .	9
2.12	Test key-spec merges . . . . .	9
2.13	Test multi-spec . . . . .	10
2.14	Test collections . . . . .	11
2.15	Test sequences . . . . .	13
2.16	Test nested regexes (regices?) . . . . .	13
2.17	Test runtime validation (:pre and :post) . . . . .	13
2.18	Test fdef [sic; not =ifdef=] . . . . .	14
2.19	Card game . . . . .	15

26	2.20 Testing test.check . . . . .	16
27	2.21 Instrumentation and Testing . . . . .	20
28	<b>3 TESTING</b>	<b>22</b>
29	3.1 Original spec reveals a bug . . . . .	23
30	3.2 Constrained spec fixes the bug . . . . .	23
31	3.3 Relaxed spec has a different bug . . . . .	24
32	<b>4 TODO: ORCHESTRA (BEYOND INSTRUMENT) AND EXPOUND (BEYOND EXPLAIN)</b>	<b>26</b>
33	<b>5 TODO: ENUMERATE NAMESPACE</b>	<b>26</b>
34	5.1 CORE . . . . .	26
35	5.2 SPEC.ALPHA . . . . .	26
36	5.3 MONADS . . . . .	26
37	<b>6 ARDES URLS</b>	<b>26</b>

## 38 1 INTRODUCTION

39 Leslie Lamport said, roughly

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

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

46 Writing a spec is an art rather than a science. A spec should constrain your system to do what it's supposed to do  
 47 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,  
 48 every system trivially satisfies a spec that always says *true*. That's a valid spec, but it's not useful, because your  
 49 system might crash or go into an infinite loop or launch the missiles, and still satisfy the spec. If your spec is too  
 50 tight, your system might not generalize well. For example, if you write a spec that requires all outputs from a  
 51 random-number generator to be positive, then you'll never get a zero or a negative random number. That may be  
 52 exactly what you want, or it may be a sloppy spec that breaks later when you need non-positive randoms or, worse,  
 53 doesn't express what you really wanted, which was non-negative randoms.

54 A *formal verification* or *certification* is a proof that your system satisfies a spec. I define a *quasi-formal* verification as  
 55 a proof that your system *probably* satisfies a spec. To do a quasi-formal verification, Clojure feeds random data into  
 56 the system and then checks the spec. Clojure uses your spec to generate random input data. You should formally  
 57 specify the domains of all your inputs. Domains are, sets, like the integers, or the floating-point numbers, or rows  
 58 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.)

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

## 97 2.2 Require the spec package

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

## 99 2.3 Test s/conform

100 (s/conform even? 1000)

101 1000

102 (s/conform even? 1001)

103 :clojure.spec.alpha/invalid

## 104 2.4 Test s/valid?

105 (s/valid? even? 1000)

106 true

107 (s/valid? even? 1001)

108 false

## 109 2.5 Test sets as predicates

110 This import works best when outside the block that follows it

111 (import java.util.Date)

112 java.util.Date

113 All the following should be true:

```
114 (every? true?  
115       [(s/valid? nil? nil)  
116        (s/valid? string? "abc")  
117  
118        (s/valid? #(> % 5) 10)  
119        (not (s/valid? #(> % 5) 0))
```

```

120
121      (s/valid? inst? (java.util.Date.))
122
123      (s/valid? #{:club :diamond :heart :spade} :club)
124      (not (s/valid? #{:club :diamond :heart :spade} 42)) ])

125 true

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

132 true

```

## 133 2.6 Test doc

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

```

135 (ns my.domain)
136 (clojure.repl/doc ::date)
137 (clojure.repl/doc ::suit)

138 -----
139 :my.domain/date
140 Spec
141   inst?
142 -----
143 :my.domain/suit
144 Spec
145   #{:spade :heart :diamond :club}

```

## 146 2.7 Test s/or

```

147 (ns my.domain)
148 ( ->> [ (s/def ::name-or-id (s/or :name string? :id int?))
149
150         (s/valid? ::name-or-id "abc")
151         (s/valid? ::name-or-id 100)
152         (not (s/valid? ::name-or-id :foo)) ]
153
154       (drop 1) (every? true?))

155 true

```

## 156 2.8 Test explain

```

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

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

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

164 #:clojure.spec.alpha{:problems
165                        ([:path [:name],
166                          :pred clojure.core/string?,
167                          :val :foo,
168                          :via [:my.domain/name-or-id],
169                          :in []]
170                          [:path [:id],
171                          :pred clojure.core/int?,
172                          :val :foo,
173                          :via [:my.domain/name-or-id],
174                          :in []])),
175                        :spec :my.domain/name-or-id,
176                        :value :foo}

```

## 177 2.9 Test Entity Maps

```

178 (ns my.domain)
179 (def email-regex #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
180 (s/def ::email-type (s/and string? #(re-matches email-regex %)))
181
182 (s/def ::acctid int?)
183 (s/def ::first-name string?)
184 (s/def ::last-name string?)
185 (s/def ::email ::email-type)
186
187 (s/def ::person (s/keys :req [::first-name ::last-name ::email]
188                          :opt [::phone]))
189 (println *ns*)

190 #namespace[my.domain]

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

196 true

```

197 I can't get the following to word wrap despite [https://www.rosettacode.org/wiki/Word\\_wrap#Clojure](https://www.rosettacode.org/wiki/Word_wrap#Clojure):

```

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

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

202 (ns my.domain)
203 (s/explain ::person
204         {:first-name "Bugs"
205          :last-name "Bunny"
206          :email "n/a"})

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

208 (ns my.domain)
209 (s/def :unq/person
210       (s/keys :req-un [:first-name :last-name :email]
211               :opt-un [:phone]))
212
213 (s/conform :unq/person
214           {:first-name "Bugs"
215            :last-name "Bunny"
216            :email "bugs@example.com"})
217 ;;=> {:first-name "Bugs", :last-name "Bunny", :email "bugs@example.com"}

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

219 (ns my.domain)
220 (s/explain :unq/person
221           {:first-name "Bugs"
222            :last-name "Bunny"
223            :email "n/a"})
224 ;; "n/a" - failed: (re-matches email-regex %) in: [:email] at: [:email]
225 ;; spec: :my.domain/email-type
226
227 (s/explain :unq/person
228           {:first-name "Bugs"})
229 ;; {:first-name "Bugs"} - failed: (contains? % :last-name) spec: :unq/person
230 ;; {:first-name "Bugs"} - failed: (contains? % :email) spec: :unq/person

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

```

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

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

241 **NOTICE** :eval never and begin\_example for the following. Do not evaluate them.

```

242 (s/def :unq/person
243   (s/keys :req-un [::first-name ::last-name ::email]
244     :opt-un [::phone]))
245
246 (s/conform :unq/person
247   {:first-name "Bugs"
248    :last-name "Bunny"
249    :email "bugs@example.com"})
250 ;;=> {:first-name "Bugs", :last-name "Bunny", :email "bugs@example.com"}

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

252 (s/explain :unq/person
253   {:first-name "Bugs"
254    :last-name "Bunny"
255    :email "n/a"})
256 ;; "n/a" - failed: (re-matches email-regex %) in: [:email] at: [:email]
257 ;; spec: :my.domain/email-type
258
259 (s/explain :unq/person
260   {:first-name "Bugs"})
261 ;; {:first-name "Bugs"} - failed: (contains? % :last-name) spec: :unq/person
262 ;; {:first-name "Bugs"} - failed: (contains? % :email) spec: :unq/person

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

```

## 266 2.10 Test records

```

267 (ns my.domain)
268 (def email-regex #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
269 (s/def ::email-type (s/and string? #(re-matches email-regex %)))
270
271 (s/def ::acctid int?)
272 (s/def ::first-name string?)
273 (s/def ::last-name string?)
274 (s/def ::email ::email-type)
275
276 (s/def ::person (s/keys :req [::first-name ::last-name ::email]
277   :opt [::phone]))
278 (println *ns*)

279 #namespace[my.domain]

```



```

280 (ns my.domain)
281 (defrecord Person [first-name last-name email phone])
282
283 (s/explain :unq/person
284           (->Person "Bugs" nil nil nil))
285 ;; nil - failed: string? in: [:last-name] at: [:last-name] spec: :my.domain/last-name
286 ;; nil - failed: string? in: [:email] at: [:email] spec: :my.domain/email-type

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

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

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

```

## 2.11 Test keyword args

```

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

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

```

## 2.12 Test key-spec merges

```

303 (ns my.domain)
304 (s/def :animal/kind string?)
305 (s/def :animal/says string?)
306 (s/def :animal/common (s/keys :req [:animal/kind :animal/says]))
307 (s/def :dog/tail? boolean?)
308 (s/def :dog/breed string?)
309 (s/def :animal/dog (s/merge :animal/common
310                           (s/keys :req [:dog/tail? :dog/breed])))
311 (println (s/valid? :animal/dog
312                {:animal/kind "dog"
313                 :animal/says "woof"
314                 :dog/tail? true
315                 :dog/breed "retriever"}))

316 true

```

317 Notice the specs above are not in the namespace.

```

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

```

```

320 -----
321 :animal/kind
322 Spec
323   string?

```

## 324 2.13 Test multi-spec

```

325 (ns my.domain)
326 (s/def :event/type keyword?)
327 (s/def :event/timestamp int?)
328 (s/def :search/url string?)
329 (s/def :error/message string?)
330 (s/def :error/code int?)
331
332 (defmulti event-type :event/type)
333 (defmethod event-type :event/search [_]
334   (s/keys :req [:event/type :event/timestamp :search/url]))
335 (defmethod event-type :event/error [_]
336   (s/keys :req [:event/type :event/timestamp :error/message :error/code]))
337
338 (s/def :event/event (s/multi-spec event-type :event/type))
339
340 (println
341   (every? true?
342     [(s/valid? :event/event
343       {:event/type :event/search
344        :event/timestamp 1463970123000
345        :search/url "https://clojure.org"})
346      (s/valid? :event/event
347        {:event/type :event/error
348         :event/timestamp 1463970123000
349         :error/message "Invalid host"
350         :error/code 500})]))
351
352 true

```

```

353 (ns my.domain)
354 (s/explain :event/event
355   {:event/type :event/restart})

```

```

356 Success!

```

```

357 (ns my.domain)
358 (s/explain :event/event
359   {:event/type :event/search
360    :search/url 200})

```

```

361 200 - failed: string? in: [:search/url] at: [:event/search :search/url] spec: :search/url
362 {:event/type :event/search, :search/url 200} - failed: (contains? % :event/timestamp) at: [:event/search] s

```

### 363 2.13.1 Open types

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

```

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

```

370 true

## 371 2.14 Test collections

### 372 2.14.1 homogeneous small: coll-of

```

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

```

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

```

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

```

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

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

```

387 (ns my.domain)
388 (s/explain ::vnum3 #{5 10 2})
389 (s/explain ::vnum3 [1 1 2])
390 (s/explain ::vnum3 [1 2 :a])
391 (s/explain ::vnum3 [1])
392 (s/explain ::vnum3 [1 1 :a])

393 #{2 5 10} - failed: vector? spec: :my.domain/vnum3
394 [1 1 2] - failed: distinct? spec: :my.domain/vnum3
395 :a - failed: number? in: [2] spec: :my.domain/vnum3
396 [1] - failed: (= 3 (count %)) spec: :my.domain/vnum3
397 [1 1 :a] - failed: distinct? spec: :my.domain/vnum3

```

### 398 2.14.2 homogeneous large: every, every-kv

399 s/\*coll-check-limit\*

400 101

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

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

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

409 42

### 410 2.14.3 heterogeneous: tuple

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

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

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

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

### 417 2.14.4 homogenous: map-of

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

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

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

## 2.15 Test sequences

```

425 (ns my.domain)
426 (s/def ::ingredient (s/cat :quantity number? :unit keyword?))
427 (s/conform ::ingredient [2 :teaspoon])
428
429 :my.domain/ingredient{:quantity 2, :unit :teaspoon}
430
431 (ns my.domain)
432 (s/explain ::ingredient [11 "peaches"])
433
434 "peaches" - failed: keyword? in: [1] at: [:unit] spec: :my.domain/ingredient
435
436 (ns my.domain)
437 (s/explain ::ingredient ["peaches"])
438
439 "peaches" - failed: number? in: [0] at: [:quantity] spec: :my.domain/ingredient

```

## 2.16 Test nested regexes (regices?)

```

437 (ns my.domain)
438 (s/def ::nested
439   (s/cat :names-kw #{:names}
440         :names      (s/spec (s/* string?))
441         :nums-kw    #{:nums}
442         :nums        (s/spec (s/* number?))))
443 (s/conform ::nested [:names ["a" "b"], :nums [1 2 3]])
444
445 :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.

```

447 (ns my.domain)
448 (defn person-name
449   [person]
450   {:pre [(s/valid? ::person person)]
451    :post [(s/valid? string? %)]})
452 (str (::first-name person) " " (::last-name person))
453 (println (person-name {:first-name "Bugs"
454                       :last-name "Bunny"
455                       :email "bugs@example.com"}))

```

Bugs Bunny

```

457 (ns my.domain)
458 (defn person-name
459   [person]
460   (let [p (s/assert ::person person)]
461     (str (::first-name p) " " (::last-name p))))
462
463 (s/check-asserts true) ;; <~~ Don't forget this; it's off by default.
464 (person-name 100)

465 class clojure.lang.ExceptionInfo class clojure.lang.ExceptionInfoExecution error - invalid arguments to my.d
466 100 - failed: map?

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

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

476 (ns my.domain)
477 (defn- set-config [prop val]
478   ;; dummy fn
479   (println "set" prop val))
480
481 (defn configure [input]
482   (let [parsed (s/conform ::config input)]
483     (if (= parsed ::s/invalid)
484       (throw (ex-info "Invalid input" (s/explain-data ::config input)))
485       (for [{prop :prop [_ val] :val} parsed]
486         (set-config (subs prop 1) ;; Strip the leading hyphen
487                     val))))))
488
489 (configure ["-server" "foo" "-verbose" true "-user" "joe"])

490 set server foo
491 set verbose true
492 set user joe

```

## 493 2.18 Test fdef [sic; not =ifdef=]

```

494 (ns my.domain)
495 (defn ranged-rand
496   "Returns random int in range start <= rand < end. Noti"
497   [start end]
498   (+ start (long (rand (- end start)))))
499

```

```

500 (s/fdef ranged-rand
501   :args (s/and (s/cat :start int? :end int?)
502               #(< (:start %) (:end %))))
503   :ret int?
504   :fn (s/and #(>= (:ret %) (-> % :args :start))
505           #(< (:ret %) (-> % :args :end))))

506 #'my.domain/ranged-randmy.domain/ranged-rand

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

509 -----
510 my.domain/ranged-rand
511 ([start end])
512 Returns random int in range start <= rand < end. Noti
513 Spec
514   args: (and (cat :start int? :end int?) (< (:start %) (:end %)))
515   ret: int?
516   fn: (and (>= (:ret %) (-> % :args :start)) (< (:ret %) (-> % :args :end)))

517 (ns my.domain)
518 (defn adder [x] #(+ x %))
519
520 (s/fdef adder
521   :args (s/cat :x number?)
522   :ret (s/fspec :args (s/cat :y number?)
523               :ret number?)
524   :fn #(<= (-> % :args :x) ((:ret %) 0)))
525
526 (clojure.repl/doc my.domain/adder)

527 -----
528 my.domain/adder
529 ([x])
530 Spec
531   args: (cat :x number?)
532   ret: (fspec :args (cat :y number?) :ret number? :fn nil)
533   fn: (= (-> % :args :x) ((:ret %) 0))

```

## 534 2.19 Card game

```

535 (ns my.domain)
536 (def suit? #{:club :diamond :heart :spade})
537 (def rank? (into #{:jack :queen :king :ace} (range 2 11)))
538 (def deck (for [suit suit? rank rank?] [rank suit]))
539
540 (s/def ::card (s/tuple rank? suit?))
541 (s/def ::hand (s/* ::card))
542

```

```

543 (s/def ::name string?)
544 (s/def ::score int?)
545 (s/def ::player (s/keys :req [::name ::score ::hand]))
546
547 (s/def ::players (s/* ::player))
548 (s/def ::deck (s/* ::card))
549 (s/def ::game (s/keys :req [::players ::deck]))
550
551 (def kenny
552   {::name "Kenny Rogers"
553     ::score 100
554     ::hand []})
555 (println (s/valid? ::player kenny))
556
557 (s/explain ::game
558   {::deck deck
559     ::players [{::name "Kenny Rogers"
560                  ::score 100
561                  ::hand [[2 :banana]]}]}))

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

```

## 564 2.20 Testing test.check

### 565 2.20.1 Basic generators

```

566 (ns my.domain)
567 (require '[clojure.spec.gen.alpha :as gen])
568 (clojure.pprint/pprint
569   [ (gen/generate (s/gen int?))
570     (gen/generate (s/gen nil?))
571     (gen/sample (s/gen string?))
572     (gen/sample (s/gen (s/cat :k keyword? :nums (s/* number?))) 5)
573     (s/exercise (s/cat :k keyword? :ns (s/* number?)) 5)
574     (gen/sample (s/gen (s/and int? #(> % 0) #(zero? (mod % 3)))))
575     ; (gen/generate (s/gen ::player)) ;; <o=-< works, but is too long
576     ; (gen/generate (s/gen ::game)) ;; <o=-<
577   ])

578 [-2010
579   nil
580   (" " " " " " "K" "dV" "xie" "5utRr8" "DU3" "0rB")
581   ([:L]
582     (:iX)
583     (:uWk/!9b)
584     (:Z?+0/f_Z)
585     (:!*D!.C5.1c.Y/saf9 -0.5 -1 1.5 3.125))
586   [([:H] {:k :H})
587     [(:W! -0.5) {:k :W!, :ns [-0.5]}]
588     [(:_e?/q -1) {:k :_e?/q, :ns [-1]}]
589     [(:c/MM -1 ##-Inf -0.5) {:k :c/MM, :ns [-1 ##-Inf -0.5]}]

```



```

590  [(:gz1.?+._B!.G?*B/aB6+ -2 1)
591   {:k :gz1.?+._B!.G?*B/aB6+, :ns [-2 1]})]
592  (9 12 33 3 60 57 15 20841 405 45)]

```

593 With fully qualified symbols everywhere:

```

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

```

```

595 -----
596 my.domain/ranged-rand
597 ([start end])
598 Returns random int in range start <= rand < end. Noti
599 Spec
600 args: (and (cat :start int? :end int?) (< (:start %) (:end %)))
601 ret: int?
602 fn: (and (>= (:ret %) (-> % :args :start)) (< (:ret %) (-> % :args :end)))

```

```

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

```

```

607  [[(6 7) 6]
608   [(-1 0) -1]
609   [(-1 1) -1]
610   [(-8 -4) -6]
611   [(-3 -1) -2]
612   [(-1 0) -1]
613   [(-7 30) 16]
614   [(-32 2) -12]
615   [(-204 -99) -142]
616   [(-1 12) 3]]

```

## 617 2.20.2 Testing s/with-gen

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

```

620 (ns my.domain)
621 (s/def ::kws (s/and
622               keyword?
623               #(= (namespace %) "my.domain")))
624 (s/valid? ::kws :my.domain/name) ;; true
625 (gen/sample (s/gen ::kws)) ;; overwhelmingly unlikely we'll generate useful
626                               ;; keywords this way

```

```

627 :my.domain/kwstrueclass clojure.lang.ExceptionInfoError printing return val
628 Couldn't satisfy such-that predicate after 100 tries.

```

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

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

640 (:my.domain/deck
641  :my.domain/person-name
642  :my.domain/event-type
643  :my.domain/map->Person
644  :my.domain/rank?)
```

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

```
647 (ns my.domain)
648 (s/def ::kws (s/with-gen
649               (s/and keyword? #(= (namespace %) "my.domain"))
650               #(s/gen #{:my.domain/->Person :my.domain/rank? :my.domain/person-name
651                       :my.domain/email-regex :my.domain/deck :my.domain/configure
652                       :my.domain/-syms :my.domain/map->Person :my.domain/adder
653                       :my.domain/kenny :my.domain/ranged-rand :my.domain/event-type
654                       :my.domain/kw-gen :my.domain/suit?}
655                       )))
656 (clojure.pprint/pprint
657   (gen/sample (s/gen ::kws) 5))

658 (:my.domain/rank?
659  :my.domain/person-name
660  :my.domain/-syms
661  :my.domain/rank?
662  :my.domain/suit?)
```

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

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

671 (:my.domain/->Person
```

```

672 :my.domain/kenny
673 :my.domain/person-name
674 :my.domain/suit?
675 :my.domain/deck)

```

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

```

677 (ns my.domain)
678 (def -kwds (into #{} (map #(keyword "my.domain" (str %))
679                           (keys (ns-publics 'my.domain)))))
680 (def kw-gen-2 (s/gen -kwds))
681 (s/def
682   ::kws
683   (s/with-gen
684     (s/and keyword? #(= (namespace %) "my.domain"))
685     (fn [] kw-gen-2)))
686 (clojure.pprint/pprint (gen/sample (s/gen ::kws) 5))

687 (:my.domain/kenny
688   :my.domain/deck
689   :my.domain/configure
690   :my.domain/email-regex
691   :my.domain/-kwds)

```

### 692 2.20.3 Open generator spaces with fmap

```

693 (ns my.domain)
694 (let [digit? (set (range 0 10))
695       ascint #(- (int %) 48)]
696   (clojure.pprint/pprint
697     ( ->>
698       (gen/string-alphanumeric)
699       (gen/such-that
700         #(and (not= % "")
701               (not (digit? (ascint (first %)))))
702       (gen/fmap #(keyword "my.domain" %)
703                 gen/sample)))

704 (:my.domain/0
705   :my.domain/C
706   :my.domain/H8
707   :my.domain/On2
708   :my.domain/AaYy
709   :my.domain/FFCCQA
710   :my.domain/Zn8s
711   :my.domain/zw
712   :my.domain/h75VT35m
713   :my.domain/w3Jxw)

```

```

714 (ns my.domain)
715 (s/def ::hello

```

```

716 (s/with-gen
717   #(clojure.string/includes? % "hello")
718   #(gen/fmap (fn [[s1 s2]] (str s1 "hello" s2))
719             (gen/tuple (gen/string-alphanumeric)
720                       (gen/string-alphanumeric))))
721 (clojure.pprint/pprint
722  (gen/sample (s/gen ::hello)))

```

```

723 ("hello"
724  "hello"
725  "hellooD"
726  "L54hello4b"
727  "hellosRwx"
728  "39VGshello"
729  "Xn0hello79a00"
730  "I0471oChellom"
731  "Qbhello"
732  "50hellonEer01")

```

## 733 2.20.4 Range specs and generators

```

734 (ns my.domain)
735 (-> (s/int-in 0 11)
736      s/gen
737      gen/sample)

```

```

738                                0  1  2  1  3  1  4  4  1  3

```

```

739 (ns my.domain)
740 (-> (s/inst-in #inst "2000" #inst "2010")
741      s/gen
742      (gen/sample 55)
743      ((partial take-last 5))
744      clojure.pprint/pprint
745 )
746

```

```

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

```

## 752 2.21 Instrumentation and Testing

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

$$\text{rr}(s, e) = s + \text{rand}(e - s) \quad (1)$$

754 where

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

755 and `rand([0..1])` means *a random number between 0, inclusive, and 1, exclusive*.

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

```
758 (ns my.domain)
759 (defn ranged-rand
760   "Returns random int in range start <= rand < end. Noti"
761   [start end]
762   (+ start (long (rand (- end start)))))
763
764 (s/def ranged-rand
765   :args (s/and (s/cat :start int? :end int?)
766                 #(not (neg? (:start %))) #(not (neg? (:end %)))
767                 #(< (:start %) (:end %)))
768   :ret int?
769   :fn (s/and #(>= (:ret %) (-> % :args :start))
770              #(< (:ret %) (-> % :args :end))))
```

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

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

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

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

```
784 (ns my.domain)
785 (stest/unstrument 'ranged-rand)
786 (-> (ranged-rand 8 5)
787     clojure.pprint/pprint)
788 (-> (ranged-rand -42 0)
789     clojure.pprint/pprint)
```

```
790 7
791 -11
```

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

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

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

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

```
806 (ns my.domain)
807 (defn ranged-rand
808   "Returns random int in range start <= rand < end. Noti"
809   [start end]
810   (+ start (long (rand (- end start)))))
811
812 (s/def ranged-rand
813   :args (s/cat :start int? :end int?)
814   :ret int?
815   :fn (s/and #(>= (:ret %) (-> % :args :start))
816              #(< (:ret %) (-> % :args :end))))
817
818 (-> 'ranged-rand
819     s/exercise-fn
820     clojure.pprint/pprint)
```

```
821 [(0 -1) 0]
822 [(-1 0) -1]
823 [(-1 -1) -1]
824 [(-2 -2) -2]
825 [(2 3) 2]
826 [(-2 1) 0]
827 [(-24 -1) -7]
828 [(-59 -16) -35]
829 [(-15 13) -7]
830 [(0 0) 0])
```

### 831 3 TESTING

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

834 It is perhaps surprising and certainly instructive that `ranged-rand` has bugs, and that writing and checking a good  
 835 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)
```

```

875 (s/fdef ranged-rand
876   :args (s/and (s/cat :start int? :end int?)
877                 ; OH YES, HERE IS THE FIX, NOT TO THE CODE, BUT TO THE SPEC
878                 #(not (neg? (:start %))) #(not (neg? (:end %))))
879                 #(< (:start %) (:end %)))
880   :ret int?
881   :fn (s/and #(>= (:ret %) (-> % :args :start))
882           #(< (:ret %) (-> % :args :end))))
883
884 (-> (stest/check 'ranged-rand
885         {:clojure.spec.test.check/opts
886          {:num-tests 100000}})
887      clojure.pprint/pprint)

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

```

### 893 3.3 Relaxed spec has a different bug

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

```

895 (ns my.domain)
896
897 (s/fdef ranged-rand
898   :args (s/and (s/cat :start int? :end int?)
899                 #(not (neg? (:start %))) #(not (neg? (:end %))))
900   :ret int?
901   :fn (s/and #(>= (:ret %) (-> % :args :start))
902           #(< (:ret %) (-> % :args :end))))
903
904 (-> (stest/check 'ranged-rand
905         {:clojure.spec.test.check/opts
906          {:num-tests 1001}})
907      first
908      stest/abbrev-result
909      :failure ::s/problems ;; <0=-< a new filter!
910      clojure.pprint/pprint)

911 [{:path [:fn],
912   :pred
913   (clojure.core/fn
914     [%]
915     (clojure.core/< (:ret %) (clojure.core/-> % :args :end))),
916   :val {:args {:start 0, :end 0}, :ret 0},
917   :via [],
918   :in []}]

```

919 We see that, although the return value is sensible when `start` equals `end`, it's out of spec and not very useful. Put  
920 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`.

```

924 (ns my.domain)
925
926 (s/fdef ranged-rand
927   :args (s/and (s/cat :start int? :end int?)
928                 #(not (neg? (:start %))) #(not (neg? (:end %)))
929                 #(not= (:start %) (:end %))))
930   :ret int?
931
932   :fn (s/or :regular-branch
933         (s/and
934           #(< (-> % :args :start) (-> % :args :end))
935           #(>= (:ret %) (-> % :args :start))
936           #(< (:ret %) (-> % :args :end)))
937         :reversed-branch
938         (s/and
939           #(> (-> % :args :start) (-> % :args :end))
940           #(<= (:ret %) (-> % :args :start))
941           #(> (:ret %) (-> % :args :end)))
942       ))
943
944 (-> (stest/check 'ranged-rand
945              {:clojure.spec.test.check/opts
946               {:num-tests 1001}})
947      first
948      clojure.pprint/pprint)
949
950 {:spec
951  #object[clojure.spec.alpha$fspec_impl$reify__2524 0x73ee284 "clojure.spec.alpha$fspec_impl$reify__2524@73e
952  :clojure.spec.test.check/ret
953  {:result true, :num-tests 1001, :seed 1562720011652},
954  :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’ing 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’ing 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.

## 966 4 TODO: ORCHESTRA (BEYOND INSTRUMENT) AND EXPOUND 967 (BEYOND EXPLAIN)

## 968 5 TODO: ENUMERATE NAMESPACE

### 969 5.1 CORE

```
970 (ns my.domain)
971 (-> (stest/enumerate-namespace 'clojure.core)
972      stest/check
973      clojure.pprint/pprint )
```

```
974 ()
```

### 975 5.2 SPEC.ALPHA

```
976 (ns my.domain)
977 (-> (stest/enumerate-namespace 'my.domain)
978      ;stest/check
979      clojure.pprint/pprint )
```

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

### 986 5.3 MONADS

```
987 (ns my.domain)
988 (require '[clojure.algo.monads :as m])
989 (-> (stest/enumerate-namespace 'clojure.algo.monads)
990      stest/check
991      clojure.pprint/pprint )
```

```
992 ()
```

## 993 6 ARDES URLS

### 994 ARDES 101

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

996 **ARDES 2.0 SDK**

997 [https://w.amazon.com/bin/view/Amazon\\_Robotics/Virtual\\_Systems/Engines/ARDES/SDK2.0/](https://w.amazon.com/bin/view/Amazon_Robotics/Virtual_Systems/Engines/ARDES/SDK2.0/)

998 **ARDES AirGateway Simulation**

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

1000 **ARDES Batch Interface**

1001 <https://w.amazon.com/index.php/Amazon%20Robotics/Virtual%20Systems/Developers/ArdesBatch>

1002 **ARDES CLI Command Reference**

1003 <https://w.amazon.com/index.php/Main/ARDES/Internal/ArdesCLICommandReference>

1004 **ARDES Case Depalletizer Simulation**

1005 <https://drive.corp.amazon.com/documents/OpsSimulation/AR%20ARDES%20Case%20Depalletizer%20Simulation.docx>

1007 **ARDES Developer Onboarding**

1008 <https://w.amazon.com/bin/view/Main/ARDES/Dev/Onboarding/#HRunyourfirstlocalsimulation>

1009 **ARDES FC Rolo Simulation**

1010 <https://drive.corp.amazon.com/documents/OpsSimulation/AR%20ARDES%20FC%20Rolo%20Simulation.docx>

1011 **ARDES Internal Visualization**

1012 <https://w.amazon.com/bin/view/Main/ARDES/Internal#HVisualization>

1013 **ARDES Parallel Event Coordinator**

1014 <https://w.amazon.com/index.php/Amazon%20Robotics/Virtual%20Systems/Developers/ParallelEventProcessing>

1015 **ARDES Quick Start for Mac**

1016 <https://w.amazon.com/bin/view/Main/ARDES/demo/>

1017 **ARDES ROLO (Restowing of Relocated Inventory)**

1018 [https://w.amazon.com/bin/view/Amazon\\_Robotics/Virtual\\_Systems/Engines/ARDES/ROLO/](https://w.amazon.com/bin/view/Amazon_Robotics/Virtual_Systems/Engines/ARDES/ROLO/)

1019 **ARDES SortCenter Simulation**

1020 <https://drive.corp.amazon.com/documents/OpsSimulation/AR%20ARDES%20SortCenter%20Simulation.docx>

1021 **ARDES Streaming Service**

1022 [https://code.amazon.com/packages/ARDESStreamingServiceService/blobs/mainline/--/install\\_ARDESStreamingServiceService](https://code.amazon.com/packages/ARDESStreamingServiceService/blobs/mainline/--/install_ARDESStreamingServiceService/workspace.sh?raw=1)  
1023 [workspace.sh?raw=1](https://code.amazon.com/packages/ARDESStreamingServiceService/blobs/mainline/--/install_ARDESStreamingServiceService/workspace.sh?raw=1)

1024 **ARDES Time Warp**

1025 [https://w.amazon.com/bin/view/Amazon\\_Robotics/Virtual\\_Systems/Engines/Interns/TimeWarp/](https://w.amazon.com/bin/view/Amazon_Robotics/Virtual_Systems/Engines/Interns/TimeWarp/)

1026 **Black Caiman**

1027 [https://w.amazon.com/bin/view/Black\\_Caiman/](https://w.amazon.com/bin/view/Black_Caiman/)

1028 **Study of FlexSim / ARDES integration**

1029 <https://w.amazon.com/index.php/Amazon%20Robotics/Virtual%20Systems/Developers/ArdesFlexSimIntegration>