*Proceedings of the 2014 Winter Simulation Conference*
*A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, eds.*

# HELMET: A CLOJURE-BASED RULES ENGINE FOR STOCHASTIC DEMAND SAMPLING IN ARMY FORCE STRUCTURE ANALYSIS

Thomas Spoon

Operations Research Systems Analyst
Center for Army Analysis
Ft Belvoir, VA 22060, USA

## ABSTRACT

Designing an Army force structure - the set of equipment, personnel, and skills that define the US Army - consists of a daunting set of interacting problems. Such analyses must deal with a wide range of force structure decisions, uncertainty about the future, and account for dynamics between force structure decisions. Recent methodologies at CAA use random variables for Army force structure demands. Due to constraints and dependencies, the business rules for determining a valid demand signal require more than simple draws from canonical distributions. Further, the rule-set must be open to extension to incorporate evolving sponsor constraints. Helmet is a novel Domain Specific Language (DSL) for defining complex demand sample generators. Implemented in the Clojure programming language, Helmet provides a robust, extensible platform for building stochastic force structure demands.

## 1 ARMY FORCE STRUCTURE ANALYSIS

In response to statutory and strategic requirements, the US Army regularly analyzes the shape and size of its force structure. The fundamental questions in force structure analysis are "What capabilities should the Army have?" and "How should the Army allocate resources across capabilities?" These questions are dependent on a legion of contextual assumptions including - but not limited to - budget constraints, end-strength constraints, the National Security Strategy (NSS), and regional stability. Many of the most sensitive assumptions are captured in demands for Army resources. Directed by the NSS and Army doctrine, planners determine plausible Army utilization relative to a strategic context, yielding a demand future – a set of force structure requirements defined over time. The Army uses demand futures to evaluate force structure options. At whole-Army scale, institutional processes like Total Army Analysis (TAA) serve as constraint satisfaction problems where the Army searches for a set of force structure allocations that minimize risk across one or more demand futures.

### 1.1 Demand Futures

A majority of force structure analysis processes exist to transform strategic plans and assumptions into a demand future. The two primary ingredients for any demand future are a set of primitive force structure demands and a timing function. The timing function defines how the primitive force structure demands are arranged over a time horizon. The primitives are akin to building blocks: when valid demands are arranged according to a valid timing, they form a valid demand future.

Force structure analysis employs valid demand futures, where validity is tied to the NSS and Army doctrine. Timings are typically derived from strategic planning and security constructs, and inherit the validity of the planning sources. Primitive demands acquire validity from the methodology used to produce them: multiple sophisticated processes in the force structure analytic ecosystem convey domain-specific assumptions, constraints, and findings via force structure demands. Primitive force structure demands are

distilled from high-fidelity, large-scale, and complex campaign analysis driven by Army doctrine, combat models, and Rules of Allocation for support structure. Campaign-level, or "surge" demands are married with a relatively simpler set of smaller demands – vignettes – derived from the Support to Strategic Analysis (SSA) process. Significant effort is spent ensuring that each primitive demand provides a doctrinally sound representation of the actual demand for Army force structure. Of the two ingredients, primitive demands take far more effort to develop and validate due to the possible intricacies involved. Consequently, validated primitive demands often become re-usable resources for building demand futures, or starting points for creating new primitive demands.

### 1.2 Force Structure Analysis With Stochastic Demand Futures

Recent force structure analysis methodologies focused on expanding the set of demand futures incorporated in canonical force structure analyses like TAA. Since force structure allocations are dependent on demand expectations, demand variation is a straightforward way to generate points in the space of force structure allocations. One simple strategy for exploring the space is to use many demand futures to generate candidate force structure solutions (a portfolio).

One effective way to define a variable demand future is to use a stochastic timing function to schedule primitive demands. Leveraging existing validated primitive demands, demand futures are generated by arranging the building-blocks of demand in novel configurations. The resulting futures retain implicit validity derived from the primitives. Since analysis based on demand variation is decoupled from institutional processes that build demand futures on an annual schedule, re-use of validated demand primitives also mitigates institutional latency and allows a wider range of analysis using the same source material.

### 1.3 Generating a Stochastic Demand Future

Generating demand futures requires two fundamental inputs: a corpus of primitive demands to draw from, and a set of rules for sampling from the corpus. Demand events that comprise a demand future are – at a minimum – simple records of data with a start time and a duration. Sampling rules usually come from the sponsor as business rules, or natural language descriptions of the properties inherent in a "valid" demand future. These business rules determine the stochastic properties of a demand future. For example, a notional valid demand future

- contains 3 instances of either *bar1*, *bar2*, or *bar3*, with start times drawn uniformly between 1 and 200.
- contains 2 instances of either *cat* or *bill*, where *cat* is 5 times less likely to be drawn than *bill*, start times drawn uniformly between 50 and 700, and durations drawn from a normal distribution with a mean 100 and standard deviation of 20.
- contains 1 instance of *qux*.

As depicted in Table 1, a set of demand records serves as a corpus of demand events that may comprise a demand future. Given the notional business rules above and the corpus in Table 1, we may draw a corresponding set of demand events to as depicted in Table 2.

### 1.4 Challenges Related to Stochastic Demand Futures

Early forms of stochastic demand generation in the RANGER-IPOD study used the aforementioned simplistic stochastic demand generation approach (Helms and Stoll, 2013). The RANGER-IPOD methodology assumed independence between demand events, and focused on variable event timings. Timing functions ranged from randomizing the start times of demands in a reference TAA demand future, to building qualitatively different futures based on historical models. Dependencies and existential relations between demand events were intentionally assumed away, expanding the set of valid futures.

Table 1: A corpus of demand data.

| Name | Start | Duration |
|---|---|---|
| bar1 | 10 | 1 |
| bar2 | 11 | 10 |
| bar3 | 21 | 5 |
| bill | 30 | 30 |
| cat | 2 | 1 |
| qux | 50 | 1000 |

Table 2: A stochastic demand future.

| Name | Start | Duration |
|---|---|---|
| bar1 | 197 | 1 |
| bar3 | 104 | 5 |
| bar1 | 42 | 1 |
| bill | 455 | 103 |
| bill | 332 | 87 |
| qux | 50 | 1000 |

As RANGER-IPOD gained traction, the methodology faced several business rule refinements from the sponsor. Demand events became subject to conditional dependencies, including causal or existential relationships, along with the possibility of higher-priority events truncating or otherwise transforming nearby, lower-priority events. These additional rules redefined "valid" futures, and constrained the feasible set of demand futures. Per the previous example, new complicating rules concerning concurrency, mutual exclusion, and collision could be:

- *cat* and *bar1* never occur concurrently.
- *bar2* and *cat* never occur in the same future.
- if *bar1* and *bar2* occur too close together, they should merge into an instance of *bar3*.
- if *bar3* and *bill* occur too close together, they should merge into an instance of *cat*.

While convenient, the initial simplistic approach simply fell apart under the weight of the practical concerns presented by the sponsors, and the need for an extensible, formal method of specifying stochastic demand future generation rules emerged. Helmet is an extensible Domain Specific Language (DSL) designed to express complicated sampling rules and to incorporate evolving sponsor constraints to generate valid stochastic demand futures.

### 1.5 Towards a Language for Describing Sampling Rules

Well known in fields such as in Artificial Intelligence, a DSL provides a robust means to handle an "open" and constantly changing problem domain. With an extensible, data-driven interpreter that can process domain rules, the DSL simultaneously provides high-level specification of arbitrarily complicated sampling rules and the ability to extend the language with new operators or evaluation semantics. This language-oriented strategy is common in the Lisp family of programming languages, which encourage syntactic extensions to the host language. A popular method of Lisp development – language oriented programming – aims to define the problem in its simplest, and most natural domain (the DSL), then gradually extends the host language to implement the DSL.

### 1.6 Related Concepts

In *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Peter Norvig demonstrates tailored language-oriented approaches to defining and solving artificial intelligence problems using DSLs (Norvig 1992). Much of the implementation and ideas for Helmet can be traced to Norvig's excellent Common Lisp examples. *Structure and Interpretation of Computer Programs* masterfully demonstrates language-oriented programming and DSLs using the Scheme dialect of Lisp (Abelson and Sussman, 1996). John Bentley also promotes the idea of DSLs (or "little languages") in *Programming Pearls* (Bentley 2000).

Helmet's notion of a "sample-graph" (defined later) is very similar to the notion of a "scene graph" in computer graphics. In a scene graph, a tree of nodes related to rendering, graphics transforms, primitive shapes, colors, and other graphical descriptors of a three dimensional scene, encode an expression that –

upon evaluation – results in rendering and image. The sample-graph mentioned later serves much the same purpose, with the evaluated result being a collection of samples rather than pixels.

The choice of sampling using function composition is inspired by the elegant notion of "behaviors" in Functional Reactive Programming (Hudak and Elliot, 1997). Behaviors in the early Functional Reactive Animation (FRAN) system are functions that map a context – typically time – to a varying output (Hudak and Elliot, 1997). Using combinators (higher order functions that compose behaviors) , the FRAN library provides an elegant means for declaring time-varying computations, specifically graphical animation and user interactions. Sampling a reactive behavior consists of applying the behavior to a point in time.

## 2 HELMET OVERVIEW

Helmet is a simple language where expressions correspond to a computation that produces a vector of samples. The language is codified as a set of functions that map expressions in the DSL – reified as Clojure expressions and literals – to a corresponding sample-graph. When a collection of samples are desired from a corpus of primitive samples, the sample-graph is passed to an evaluator function along with the corpus of primitive samples (the sampling context). The evaluator enforces the semantics of each node in the sample-graph, traversing each node of the graph until reaching a primitive sampling rule. Traversing the graph effectively transforms the primitive sampling rule according to the "meaning" of the path, returning an interpreted result (a sample).

The idea is to allow users to build an abstract syntax tree using a simple and extensible language, which describes a potentially highly complex rule for sampling from a corpus of records. As new use cases and business rules emerge, new linguistic expressions – in the form of node types and functions for combining them – are added to the language. The leverage afforded by this approach is significant: earlier expressions (sampling rules) may be retained without any loss of backwards compatibility, novel sponsor-requirements may be codified as targeted language extensions, and the same rule set may be used with any compatible sampling corpus.

## 3 LANGUAGE SPECIFICATION

The language for describing sampling rules is fairly simple, and the semantics of evaluation are similarly straightforward. Structurally, the language comports to the following grammar:

$$Rule \Rightarrow Op\ Rules |\ Atom$$

$$Rules \Rightarrow [Rule*]$$

$$Op \Rightarrow Symb \in \{Chain\ Concatenate\ Transform\ Replications\ Choice\ Constrain\} \cup OpTable$$

$$Atom \Rightarrow Number\ |Keyword\ |String$$

$$Number \Rightarrow Int*$$

$$Int \Rightarrow n \in \{0,1,2,3,4,5,6,7,8,9\}$$

$$Letter \Rightarrow s \in \{a,b,c,...z\}$$

$$Character \Rightarrow Int\ |\ Letter$$

$$String \Rightarrow "+Character*+"$$

$$Keyword \Rightarrow :+\ Character*$$

A rule is either a combination of operations applied to sequences of rules, or an atom. *Atom*s conform to data that is self-evaluating, or constant expressions, including numbers, strings, and keywords. Keywords are symbols denoted with a : before a string, like `:this-is-a-keyword`. We include keywords because they are a feature relative to the host language, and are used as efficient representations of constant data.

Operations are defined by an extensible set of core *Op* symbols, which may be supplemented with an *OpTable* as new operations are added to the language, and one or more rules that the operation applies to. Structural nesting of operations is allowed by the grammar to enable complex expressions of sampling rules.

## 4 REPRESENTATION

### 4.1 Rules

Helmet *Rule* expressions are represented as nested map data structures. Nodes in a sample-graph are simply Clojure hash-maps, with keys for `:node-type` and `:node-data` . The associated values store a corresponding *Op*, and any supplemental data needed for sampling. Each *Op* maps to a unique `:node-type` value for each *Op* in the language. We define functions that build nodes for us, as well as a set of combinator functions that enable composing nodes. The resulting nested map represents the abstract syntax tree of the rule. Implementing new node-types, i.e. extending the set of *Op*s, means defining new constructors for the node-type and implementing its semantic behavior. The following functions define node constructors:

A generic node container:

```
(defn ->node  [type data]
  {:node-type type :node-data data})
```

A node chaining child nodes sequentially:

```
(defn ->chain [nodes]
  (->node :chain {:children nodes}))
```

A node for replicating child nodes n times:

```
(defn ->replications [n nodes]
  (->node :replications {:reps n :children nodes}))
```

A node that randomly chooses a child node:

```
(defn ->choice [nodes]
  (->node :choice {:children nodes}))
```

A node that transforms child nodes by applying function f:

```
(defn ->transform  [f nodes]
  (->node :transform {:f  f :children nodes}))
```

A node that concatenates the samples of all child nodes:

```
(defn ->concatenate  [nodes]
  (->node :concatenate {:children nodes}))
```

A node that defines a set of constraints for all child node evaluations:

```
(defn ->constrain [constraints nodes]
  (->node :constrain {:constraints constraints :children nodes}))
```

Under this simple representation, we can use Clojure functions to declaratively build sample-graphs. The following expression defines a sample-graph that evaluates a choice between two rules, `:cat` and `:qux`, repeating the choice 3 times and concatenating the resulting samples:

```
(def simple-sampler
    (->concatenate
        (->replications 3
             (->choice [:cat :qux]))))
```

### 4.2 Corpus

We further exploit the persistent map data structure by storing the sample corpus as another map, with a set of *Atom*s mapped to datums eligible for sampling. Our earlier tabular sample data becomes:

```
(def c1
  {:bar1  {:name "bar1" :start 10 :duration 1}
   :bar2  {:name "bar2" :start 11 :duration 10}
   :bar3  {:name "bar3" :start 21 :duration 5}
   :bill  {:name "bill" :start 30 :duration 30}
   :cat   {:name "cat"  :start 2  :duration 1}
   :qux   {:name "qux"  :start 50 :duration 1000}})
```

Where the nested map defined by `simple-sampler` is a tree, the map defined by `c1` is a forest of simple *Rule*s. Each rule maps to exactly one datum from the corpus, or one sample, where the sample is not a node. Using this representation, both the sample-graph and the corpus of data are unified under the same grammar that define sampling rules.

### 4.3 Sampling Contexts

When merged together, a forest of rules and a corpus are referred to as the Sampling Context. The Sampling Context is effectively *Rule* database. We can define a Sampling Context by associating a rule with `simple-sampler` in the corpus:

```
(def example-sampling-context (assoc corpus :simple-sampler simple-sampler))
```

Since we use hash-maps to build our contexts, and maps are easy to compose, both our sampling rules and the sampling corpus may be defined from partially-defined pieces that are merged together as needed. The following example builds a sampling context piecemeal:

`:bar`, `:baz`, and `:foo` name compound Rules that compose primitive rules from `c1`:

```
(def c2 {:bar (->chain  [:bar1 :bar2 :bar3])
         :baz (->choice [:bill
                         (->transform
                              {:start    (stats/exponential-dist 10)
                               :duration (stats/exponential-dist 2000)}
                         :cat)])
         :foo (->transform  {:start (stats/normal-dist 10 1)}
                 (->choice  {:bar (/ 1 3)
                             :baz (/ 1 3)
                             :qux (/ 1 3)}))})
```

`:case1` defines a rule that composes our higher-order rules `:foo`, `:bar`, and `:baz`:

```
(def c3 {:case1 (->concatenate
                  [(->replications 2   :foo)
                   (->replications 10  :qux)
                   (->replications 3   :baz)])})
```

`:sample` defines a rule that constrains the results of `:case1` and replicates `:case1` three times:

```
(def c4 {:sample (->replications 3 [(->constrain {:tfinal 5000
                                                  :duration-max 5000}
                                                  :case1)])})
```

We can merge `c1`..`c4` into a single hash-map of rules called `sample-graph`:

```
(def sample-graph (merge c1 c2 c3 c4))
```

Assuming we know how to sample from our sampling context, using a like-named function, we can attain a sample from the sample-graph using the `:case1` sampling rule:

```
(sample-from sample-graph :case1)
```

Since the sampling rule is a tree structure, it is easy to visualize and debug. We can even pipe it into an interactive graph-based program and allow users to visually define their own sampling rules, or, as previously shown, define rules in Clojure code.

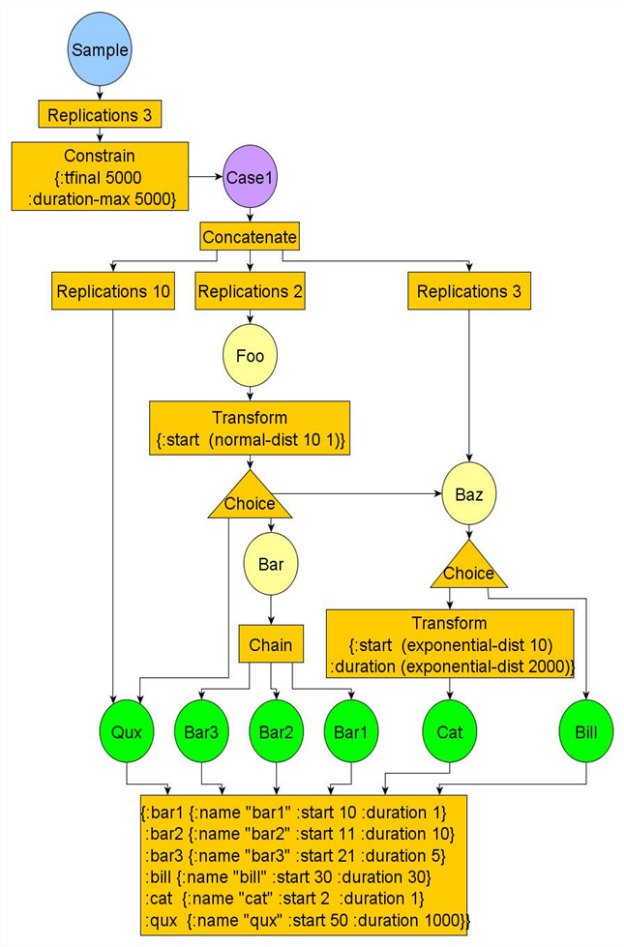


Figure 1: A visualization of `sample-graph` rooted at `:sample`.

## 5 SAMPLING

### 5.1 Implementing Sampling Semantics in Sampling Contexts

While the map-based representation solves the problem of encoding a sampling rule, generating samples from a corpus using a sampling rule requires applying the semantics of the DSL to the corpus. To actually derive a sample, we define a function that evaluates the sampling rule in the "context" of the corpus. The rule structure is traversed in order from an initial rule specified by the caller, and the leaves of the traversal result in a sampling action. The sampling action at a particular leaf node is declaratively defined by the path of nodes leading down to the leaf node. So we have a simple tree-walker that visits the leaf nodes, applies the rule to the context, and returns all of the samples.

This strategy defines an interpreter – or an evaluator in Lisp parlance – that knows how to traverse the implied AST. Lisp typically defines an `eval` function, which maps a context and an expression to be

evaluated to an evaluated result. `eval` then recursively traverses the expression, modifying and passing the context to dependent evaluation stages.

Rather than the more generally-named `eval`, `sample-node` will define an interpreter of sampling rules (Helmet expressions) in a sampling context. Since Helmet is designed to be extensible, it spreads the definition of `sample-node` across multiple implementations that correspond to different Ops. `sample-node` is implemented as a Clojure multimethod, allowing new implementations of `sample-node` as new Ops (and corresponding nodes) are added to the language. This provides a means to extend the interpreter in small, controlled amounts, without breaking the semantics of previously defined Ops. For brevity, the implementation of `sample-node` is elided in this paper. The reader is encouraged to examine the full source at https://github.com/joinr/helmet.

### 5.2 Implementing Ops With Samplers

Samplers are functions that map a sampling context to a sample. Many samplers are directly associated with Ops, since they implement the sampling behavior described by the DSL and its operations. Some samplers play a lower-level supporting role, and serve to ease implementation details. As combinators, samplers define mechanisms for composing node behaviors under a given sampling context. Sampler combination always yields another sampler. The combinators enforce the semantics of different node types, and yield new contextual functions, establishing a means for composing node semantics in any sampling context.

Deriving samples from a sampling context using a sampler is basic function application. Assuming we have a sampler, `get-samples`, we can apply the sampler to a sampling context to generate a sample.

```
(def get-sample (fn [ctx] (get ctx :blah)))
(get-sample {:blah {:name "blah" :data 99999}})
;=>{:name "blah" :data 99999}
```

Sometimes, temporarily altering the sampling context for samplers "downstream" is useful. `merge-context` samples by merging the value of a hash-map into the sampling context prior to sampling, effectively altering the sampling context.

```
(defn merge-context  [rec nodes]
  (fn [ctx] (nodes (merge rec ctx))))
```

`compose` returns a sampler that samples `n1` with a context, then samples `n2` with the resulting context of `n1`'s sampling, where `n1` and `n2`are both samplers. `compose` acts like function composition, except with samplers.

```
(defn compose  [n1 n2]
  (fn [ctx] (n2 (n1 ctx))))
```

`choice` takes a list of node samplers, and creates a node sampler that will randomly sample using a sampler from the list. The probability of selecting a node is even, unless a map of probabilities are provided.

```
(defn choice
  ([sample-func nodes]
    (fn [ctx] ((sample-func nodes) ctx)))
  ([nodes] (choice sample-nth nodes)))
```

`weighted-choice` acts like choice, except it takes a map of node → probability densities. Where the keys are resolvable nodes, and the densities are the probabilities from [0 1] that a node will be chosen. Densities must sum to unity to be valid.

```
(defn weighted-choice  [pdf-map]
  (assert (= 1.0 (float (reduce + (vals pdf-map))))
          (str "Probability densities must sum to 1.0"))
  (let [nodes  (keys pdf-map)
```

```
      choose (fn [] (loop [r (stats/*rand*)
                           xs node
                           ds (vals pdf-map)]
                      (cond (= (count xs) 1)  (first xs)
                            (<= r (first ds)) (first xs)
                            :else (recur (- r (first ds))
                                         (rest xs) (rest ds))))))]
    (fn [ctx] ((choose) ctx))))
```

`concatenate` takes an arbitrary number of samplers, applies them to a sampling context, and concatenates their samples into a vector.

```
(defn concatenate [nodes]
  (fn [ctx] (reduce conj [] (map (fn [f] (f ctx)) nodes))))
```

`replications` takes a positive integer, `n`, and performs `n` independent applications of the samplers defined by nodes, implicitly concatenating the results.

```
(defn replications [n nodes]
  (let [rep (fn [ctx f]  (reduce conj [] (map (fn [i] (f ctx)) (range n))))]
    (fn [ctx]
      (vec (map (partial rep ctx) (if (sequential? nodes) nodes [nodes]))))))
```

`transform` returns a sampler that maps `f` the result of a child sampler applied to the sampling context.

```
(defn transform  [f node]
  (fn [ctx] (f (node ctx))))
```

Given a hash-map of values, `merge-record` transforms each sampler by merging the record with the sampled result. `merge-record` is useful for defining templated transforms based on a hash-map, which are then applied to large swaths of samples.

```
(defn merge-record [rec nodes]
  (transform #(merge % rec) nodes))
```

`chain` creates a node that aggregates a list of nodes by sampling them and then "chaining" the samples according to a function. The default behavior chains nodes by arranging adjacent nodes to "follow" each other temporally, so their start and end times overlap like links in a chain.

```
(defn chain
  ([chain-func nodes]
    (fn [ctx]
      (chain-func (map (fn [nd] (nd ctx)) nodes))))
  ([nodes] (chain following-recs nodes)))
```

`with-constraints` uses a hash-map of constraint data to filter the result of any samples. Typical constraints are an upper bound on the events in the future, an upper bound on the duration for each event, and a random-number seed.

```
(defn with-constraints  [{:keys [tstart tfinal duration-max seed] :as constraint-map}
   nodes]
  (let [tstart       (or tstart 0)
        tfinal       (or tfinal Double/POSITIVE_INFINITY)
        duration-max (or duration-max Double/POSITIVE_INFINITY)
        seed         (or seed (rand-int Integer/MAX_VALUE))
        global-bounds [tstart tfinal]
        constrain    (if (unbounded-segment?  global-bounds)
                       identity
                       #(map (partial truncate-record
                                      (segment->record global-bounds)) %))]
    (fn [ctx]
```

```
(->> (stats/with-seed seed (flatten (nodes ctx)))
     (constrain)
     (filter (complement nil?))))))
```

## 6 EXAMPLE

We set an upper bound for our simulated demand future at 1500 days:

```
(def ubound 1500)
```

We can sample random times in our future drawing from a uniform distribution between 1 and 1500:

```
(def get-random-time (stats/uniform-dist 1 ubound))
```

We have a primitive corpus of sampling rules that we use to develop arbitrarily complex sampling rules for building demand futures. These are the conceptual building blocks for generating stochastic futures: *Big* demands last about 3 years and requires 1000 items. *Medium* demands last for a year, requiring 500 items. *Small* demands last for a month, requiring 100 items. *Tiny* demands are short, low-cost demands lasting only 4 days. Finally, a *year round* demand exists across the entire future, requiring a relatively modest amount of resources.

```
(def corpus
  {:Big        {:name "Big"    :start 0 :duration 1095 :quantity 1000}
   :Medium     {:name "Medium" :start 0 :duration 365  :quantity 500}
   :Small      {:name "Small"  :start 0 :duration 30   :quantity 100}
   :Sporadic   {:name "Tiny"   :start 0 :duration 4    :quantity 20}
   :Year-Round {:name "Year round demand" :start 0 :duration 1500 :quantity 15}})
```

With these building blocks, we can use our DSL to gradually express more sophisticated sampling rules: `random-surge` chooses from a *Big*, *Medium*, or *Small* demand, and places the result somewhere on the timeline. The choice will not be available for future samples, until all choices have been exhausted.

```
(def random-surge (->transform
                    {:start      (get-random-time)}
                    (->without-replacement [:Big :Small :Medium])))
```

A `ramp` demand is a sequential chain of *Small*, *Medium*, and *Big* demands that are temporally adjacent:

```
(def ramp (->chain [:Small :Medium :Big]))
```

A `random-buildup` demand randomly places `ramp` demands along the timeline, preserving the ordering and contiguiuty:

```
(def random-buildup
  (->transform
   (fn [xs] (let [offset (get-random-time)]
              (map #(update-in % [:start] + offset) xs)))  :ramp))
```

A `smallish-surge` chooses between a *Small* and a *Medium* demand, with the former being chosen 3 times as often as the latter. The chosen demand is again placed randomly on the timeline:

```
(def smallish-surge
  (->transform
   {:start    (get-random-time)}
   (->choice {:Small 3 :Medium 1})))
```

Given the plethora of partial sampling rules we defined, as well as the primitive rules in the corpus, we can compose them into yet another sampling rule, *random-case*. The new rule collects the results of executing multiple replications of random surge events, sporadic events, and small surge events. Additionally, *random-case* ensures that the *Year-Round* demand is present, as well as one *random-buildup*.

```
(def random-case (->concatenate [(->replications 2  :random-surge)
                                 (->replications 4  :smallish-surge)
                                 (->replications 10 :random-sporadic)
                                 :random-buildup
                                 :Year-Round]))
```

For clarity, we establish a sampling context by merging the corpus of primitive rules with the newly defined, complex sampling rules:

```
(def sampler (merge corpus
                    {:random-surge     random-surge
                     :ramp             ramp
                     :random-buildup   random-buildup
                     :smallish-surge   smallish-surge
                     :random-sporadic  (->transform
                                          {:start  get-random-time} :Sporadic)
                     :random-case      random-case}))
```

Finally, we can generate a random sample, in this case a notional demand future for *Small, Medium, Big, Tiny, Year-Round* demands. The function `random-sample!` takes a number of replications, `n`, and a rule to sample from, `entry-case`. Given these arguments, the function defines a constrained sampling rule that limits generated samples to the established timeline, and also limits the maximum duration of any demand to 1500 days. The in-lined constraints are then sampled `n` times to generate a set of demand records, or a stochastic demand future:

```
(defn random-sample! [n entry-case]
  (->> (->constrain {:tfinal ubound
                     :duration-max 1500}
                    [entry-case])
       (->replications n)
       (->flatten)
       (sample-from sampler)))
```

Figures 2 and 3 depict samples drawn from `(random-sample! 1 :random-case)` in a track-chart format. Demand events are arrayed horizontally by earliest start time, with concurrent events stacked vertically.

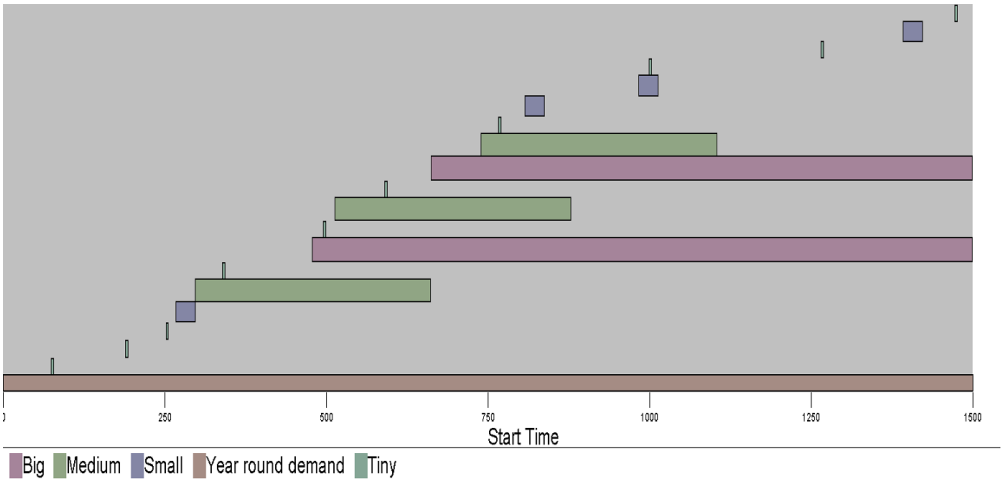


Figure 2: A Stochastic Demand Future From `random-sample!` .

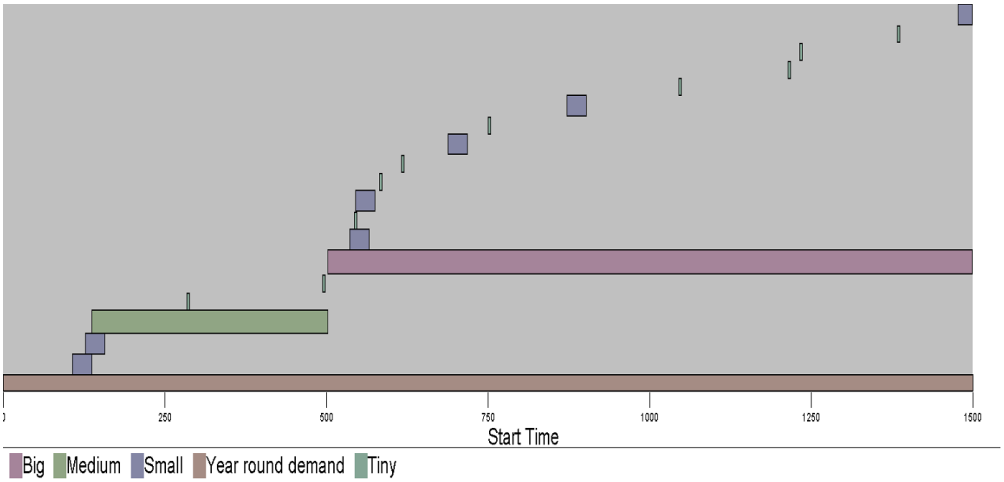


Figure 3: Another Stochastic Demand Future From `random-sample!` .

## 7 CONCLUSION

Helmet currently serves as the backbone for defining stochastic demand futures in force structure analyses at CAA. Typical usage involves reading a set of business rules from a data source, such as an Excel workbook. The rules are specified either as Clojure code, or parsed from a legacy tabular format inherited from RANGER-IPOD. Once a case is defined, demand futures are generated as needed. Most demand future generation is performed in-memory, with Helmet-based demand future streams serving as random variables for other processes like force structure analysis simulations.

When business rules are evolving or open-ended, an extensible Domain Specific Language provides a serious amount of leverage in addressing the problem domain in an elegant and robust manner. Helmet can and will keep pace with emerging business rules by extending the language, and growing with the problem domain in a controlled manner. More importantly, analysts working with stochastic demand futures can express sampling rules in a manner much closer to the original sponsor language.

## ACKNOWLEDGMENTS

Thanks to Rick Hanson, Scott Nestler, and Dwight Nwaigwe for proofing this paper.

## REFERENCES

Abelson, H., and G. J. w. J. S. Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd Editon ed. Cambridge, MA: MIT Press/McGraw-Hill.

Bentley, J. L. 2000. *Programming Pearls*. 2 ed. Reading, MA: Addison-Wesley.

Elliott, C., and P. Hudak. 1997. "Functional Reactive Animation.". In *ICFP*, edited by S. L. P. Jones, M. Tofte, and A. M. Berman, 263–273. Amsterdam, Netherlands: ACM.

Helms, J., and G. Stoll. 2012. "Randomly Generated Requirements Informed by Past Operational Deployments (RANGER IPOD)". Technical report, Center for Army Analysis.

Norvig, P. 1992. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. San Francisco, CA: Morgan-Kaufmann Publishers, Inc.

## AUTHOR BIOGRAPHIES

**THOMAS SPOON** is an Operations Research Systems Analyst for the United States Army. His email address is thomas.spoon@us.army.mil.