# Alloy as an ~~Introduction~~ Gateway to Formal Methods

Hillel Wayne

**SIMPLE**: Minimal Essential Complexity

**CONVENIENT**: Minimal Accidental Complexity

# Convenience Hypothesis

People prefer a convenient but imperfect tool over a perfect but inconvenient one.

# Alloy is convenient

# What are the “easiest” improvements?

# “Easiest” Improvements

- No centralization
- No organization
- No language extensions
- No programming

- Online Reference

## Signatures

### General

```
sig qualified-name ... {
  field declarations
}
// or ...
sig qualified-name ... {
  field declarations
}{
  signature facts
}
```

Given `sig S { ... } { F }`, *F* is interpreted as if the model read `sig S { ... } fact { all this : S | F' }`, where *F'* is like *F* but each name *f* is expanded to `this.`*f* if *f* names a field of *S*. Write `@`*f* to suppress the expansion.

### Top-level type signatures

```
sig qname { ... }
```

### Subtype signatures

```
sig qname extends superclass { ... }
```

N.B. If *A* and *B* each extend *C*, then *A* and *B* are disjoint.

### Subset signatures

```
sig qname in sup { ... }
sig qname in sup1 + sup2 + ... { ... }
```

N.B. Subset signatures are *not* necessarily pairwise disjoint, and may have multiple parents.

### Multiple signatures

```
sig qname1, qname2, ... { ... }
≡sig qname1 { ...} sig qname2 { ... }
...
```

## Paragraphs

### Facts

```
fact name { formulas }
// name is optional:
fact name { formulas }
```

### Predicates, Run

Predicates are either true or false; they can take arguments.

```
pred name { formulas }
pred name [decl, decl2 ...]{ formulas }
```

Use `run` to request an instance satisfying the predicate:

```
run name
```

Optionally specify *scope* (defaults to 3):

```
run name for 2
run name for 2 but 1 sig1, 5 sig2
```

The function `disj` is **predefined**; true iff its arguments are mutually disjoint.

### Assertions, Check

```
assert name { formulas }
```

Unlike predicates, assertions don't bind arguments.

Use `check` to look for counter-examples:

```
check name for 2 but 1 sig1, 5 sig2
```

### Functions

```
fun name [decl, ...] : e1 { e2 }
```

The body expression *e2* is evaluated to produce the function value; the bounding expression *e1* describes the set from

## Declarations

Fields of signatures, function arguments, predicate arguments, comprehension variables, quantified variables all use same declaration syntax:

### Simple declaration

```
name : bounding-expression
```

Constrains values to be a subset of the value of the bounding expression.

### Multiple declaration

```
name1, name2 : bounding-expression
// or
disj name1, name2 : bounding-expression
```

In field declarations, `disj` can also be on the right:

```
sig S {f : disj e}
```

Requires distinct *S* atoms to have distinct *f* values; ≡ `all a, b : S | a != b implies no a.f & b.f` ≡ `all disj a, b : S | disj [a.f, b.f]`

### Multiplicities

Default multiplicity is one:

```
name1 : bounding-expression
// equivalent to:
name1 : one bounding-expression
```

Other multiplicities:

```
name2 : lone expr // at most one
name3 : some expr // one or more
name4 : set expr // zero or more
```

### Relations

Bounding expression may denote a relation:

```
r : e1 -> e2
```

Multiplicities in declaring relations:

```
r : e1 -> one e2 // total function
r : e1 -> lone e2 // partial function
r : e1 one -> one e2 // 1:1 (bijection)
```

## Formulas

Formulas (aka constraints) are boolean expressions. Primitive boolean operators include the comparison operators:

```
set1 in set2
set1 = set2
scalar = value
```

Expression quantifiers make booleans out of relational expressions.

```
some relation-name
no r1 & r2 // etc.
```

Quantified expressions are formulas:

```
some var : bounding-expr | expr
all var : bounding-expr | expr
one var : bounding-expr | expr
lone var : bounding-expr | expr
no var : bounding-expr | expr
```

True iff *expr* is true for some, all, exactly one, at most one, or no elements of the set denoted by *bounding-expr*

The logical operators (`not`, `and`, `or`, `implies`, `iff`) can form compound booleans; most of them apply *only* to boolean expressions.

```
boolean and boolean2
not boolean or boolean2
boolean implies boolean2 // etc.
```

## Precedence

In precedence order.

*a*, *b*, *c* are *n*-ary relations ($n \neq 0$), *f* a functional relation, *r*, *r1*, *r2* are binary relations, *s* is a set (unary relation).

N.B. ≡ is standard mathematical syntax, not Alloy syntax.

**Unary operators:** `~r` (transpose / inverse), `^r` (positive transitive closure), `*r` (reflexive transitive closure)

**Dot join:** `a.b`

**Box join:** `b[a]` (also for function application, `f[t]`). N.B. dot binds tighter than box, so `a.b[c]` ≡ `(a.b)[c]`

**Restriction:** `s <: a` (domain restriction), `a :> s` (range restriction)

**Arrow product:** `a -> b` (Cartesian product)

**Intersection:** `a & b` (intersection*)

**Override:** `r1 ++ r2` (relational override)

**Cardinality:** `#a` (how many members in a?**)

**Union, difference:** `a + b` (union*), `a - b` (difference*)

**Expression quantifiers, multiplicities:** `no`, `some`, `lone`, `one`, `set`

**Comparison negation:** `not`, `!`

**Comparison operators:** `in`, `=`, `<`, `>`, `=`, `=<`, `=>`

**Logical negation:** `not`, `!`

**Conjunction:** `and`, `&&`

**Implication:** `implies`, `else`, `=>`

**Bi-implication:** `iff`, `<=>`

**Disjunction:** `or`, `||`

**Let, quantification operators:** `let`, `no`, `some`, `lone`, `one`, `sum`

* a and b must have matching arity

** Arithmetic overflow may occur.

## Associativity:

Implication associates right: `p => q => r` ≡ `p => (q => r)`

`else` binds to the nearest possible `implies`: `p => q => r else s` ≡ `p => (q => r else s)`

*All other binary operators* associate left: `a.b.c` ≡ `(a.b).c`

## Conditional expressions

```
boolean implies expression
boolean implies expr1 else expr2
```

## Let expressions

```
let decl, decl2 ... | expression
let decl, decl2 ... { formulas }
```

## Relational expressions

Constants: `none` (the empty set), `univ` (the universal set), `iden` (the identity function).

Compound expressions: *r1 op r2* where *op* is a **relational operator** (`->`, `.`, `[]`, `~`, `&`, `*`, `<:`, `:>`, `++`).

## Integer expressions

**Arithmetic operators** (`plus`, `minus`, `mul`, `div`, `rem`) apply only to integer expressions. They name ternary relations, so $x + 1$ can be written as any of: `plus[x][1]`, `plus[x,1]`, `x.plus[1]`, or `1.(x.plus)`.



## Module structure

```
// module declaration
module qualified/name

// imports
open other_module
open qual/name[Param] as Alias

// paragraphs (any order)
sig name ...
fact name { formulas }
pred name { formulas }
assert name { formulas }
fun name [Param] : bounding-expr {
  body-expression
}
run pred-name for scope
check assertion for scope
```

## Lexical structure

**Characters:** any ASCII character except \ ` $ % ?

Alloy is **case-sensitive**

**Tokenization:** any whitespace or punctuation separates tokens, *except* that `=>` `>=` `=<` `->` `<:` `:>` `++` `||` `//` `--` `/*` `*/` are single tokens (so: `!=` can be written `! =`)

**Comments:** from `//` to end of line; from `--` to end of line; `/*` to next `*/` (no nesting).

**Identifiers (names):** letters, numerals, underscore, quote marks (*no hyphens*)

**Qualified names (qnames):** sequence of slash-separated names, optionally beginning with `this` (e.g. `xyz`, `this/a/b/c`, `util/ordering`)

**Numeric constant:** [1-9][0-9]*

**Reserved words:** `abstract all and as assert but check disj else exactly extends fact for fun iden iff implies in Int let lone module no none not one open or pred run set sig some sum univ`

**Namespaces:** 1 module names and aliases; 2 signatures, fields, paragraphs (facts, predicates, assertions, functions), bound variables; 3 command names. Names in different namespaces do not conflict; variables are lexically scoped (inner bindings shadow outer). Otherwise, no two things can share a name.

- Online Reference
- Online Documentation

# INTRODUCTION

**Note**

*If you want a video introduction, come watch my Strange Loop talk on TLA+!*

## What is TLA+?

TLA+ is a *formal specification language*. It's a tool to design systems and algorithms, then programmatically verify that those systems don't have critical bugs. It's the software equivalent of a blueprint.

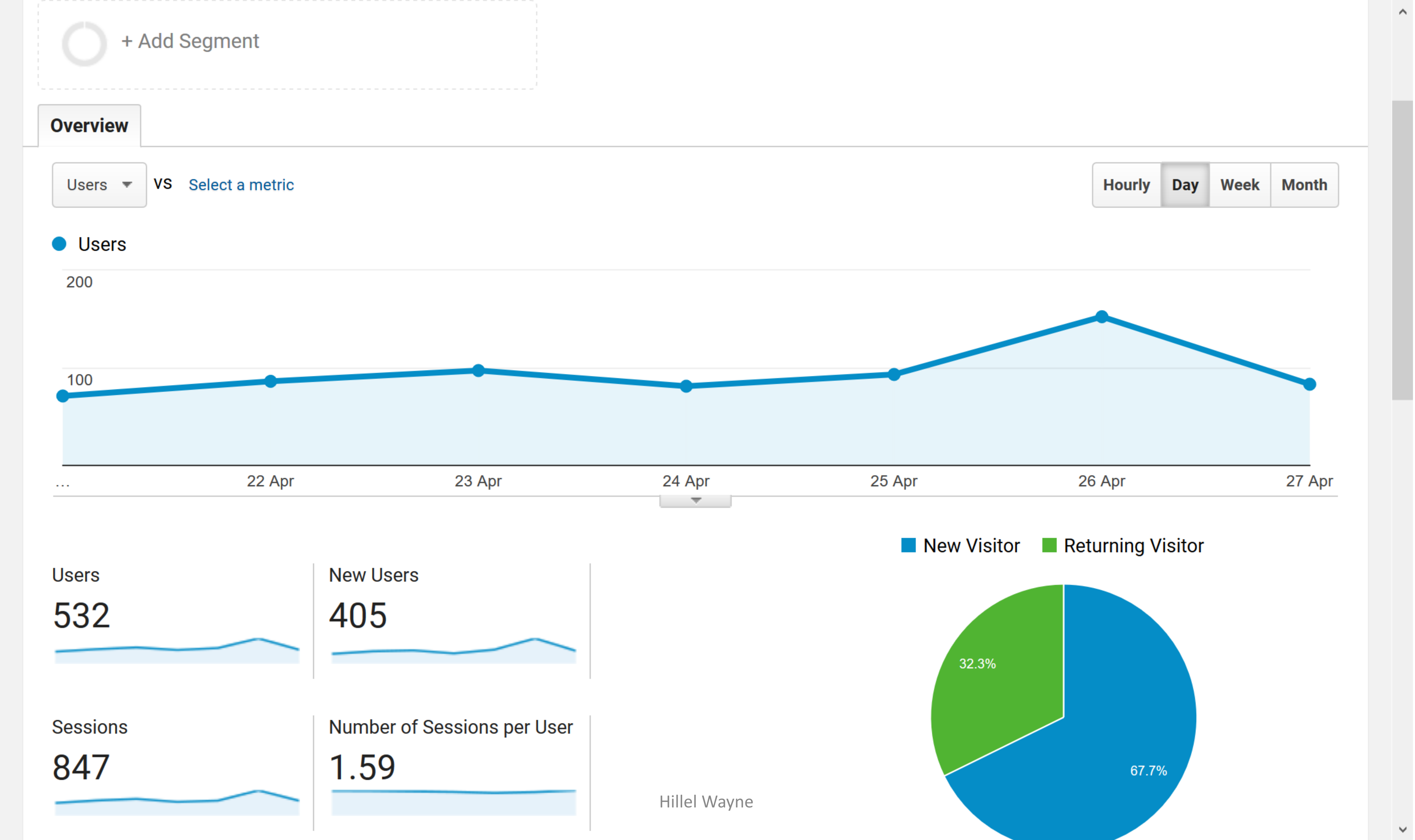
## Why should I use it?

Here's a simple TLA+ specification, representing people trading unique items. Can you find the bug?

```
People == {"alice", "bob"}
Items == {"ore", "sheep", "brick"}
(* --algorithm trade
variable owner_of \in [Items -> People]

process giveitem \in 1..3 \* up to three possible trades made
variables item \in Items,
          owner = owner_of[item],
```

Hillel Wayne




Hillel Wayne

- Online Reference
- Online Documentation
- Recipe Books

- (Identity of set) A <: iden
- (Is symmetric) ~r in r
- (Rest of cycle) a.(^succ + ^~succ)
- Traces
- Stuttering
- etc

- Online Reference
- Online Documentation
- Recipe Books
- Directed Labs

**Miikka Koskinen**
@arcatan

@Hillelogram Inspired by your blog posts, I attempted to model a work project in Alloy.

I discovered that there are huge number of invariants that we believe our data model has but that are enforced only accidentally. Now thinking about enforcing them more explicitly.

2:00 AM - Mar 23, 2018

6 See Miikka Koskinen's other Tweets