Alloy: A Quick Reference and an interpretation into B

[Marc Frappier](http://info.usherbrooke.ca/mfrappier/), 2020-11-09  
version 2.1

Inspired from the document [Alloy Quick Reference](https://www.monperrus.net/martin/alloy-quick-ref.pdf) written by [*Martin Monperrus*](http://www.monperrus.net/martin/)

<https://www.monperrus.net/martin/alloy-quick-ref.pdf>

**Alloy Specification**

The typical structure of an Alloy specification is as follows

* Declaration of signatures
* Declaration of facts
* Declaration of predicates and functions
* Run statement and check statements

However, these can be freely mixed (ie, no ordering is imposed on the declarations).

**Alloy Expressions**

* Basic types are declared using signatures.
* A signature declares a set of atoms.
* An expression is either a term or a formula.
* A type can be a signature or a term constructed using signatures.
* A variable *v* must be typed using the declaration *v* : *T*, where *T* is a term constructed using signatures.
* Variable names can be primed ; ex: x, x’, x’’, x’’’
* Alloy terms (ie, values other formulas) are *nary*-relations.
  + Alloy has no explicit notion of sets, tuples or scalars; a term is a *nary*-relation
  + A tuple is represented using a singleton relation.
  + A scalar is represented using a singleton, unary relation
  + A set is represented using a unary relation.

**Alloy terminology (as defined in Daniel Jackson’s book *Software Abstractions : Logic, Language, and Analysis*)**

* A **model** is an Alloy specification
* A **fact** is a formula that must be satisfied by a model instance
* A **model instance** is an assignment of values to the symbols (signature and relations) that satisfies the facts and the signature constraints of a specification.
  + This is a bit confusing wrt to the usual terminology in logic: a model in logic is what is called a model instance in Alloy.
* A **signature** is a set of atoms of the same type; a signature also denotes a type whose value is its set of atoms.
* A **field** is declared in a signature and it denotes a relation. A field may have constraints on its values (one, lone, set).
* An **atom** is an element of a signature. An atom is a unary relation with only one element (ie, a singleton set).

**Signatures**

|  |  |  |
| --- | --- | --- |
| *Notation* | *Intuitive Meaning* | *Equivalent B declaration* |
| sig Book {…} | Declares a set Book | SETS Book |
| sig Book { author: Author }  sig Author {…} | Declares a set Book, and a total function author | SETS Book, Author  CONSTANTS author  PROPERTIES author : Book --> Author |
| sig Book { author: set Author } | Declares a set Book, and a relation author which is a subset of the Cartesian product  Book × Author | …  PROPERTIES author : Book <-> Author |
| sig Book { author: some Author } | a book has at least one author | PROPERTIES  author : Book <-> Author  dom(author) = Book |
| sig A { f: lone B } | f is a partial function from A to B | f : A +-> B |
| sig A { f: B } | f is a total function from A to B | f : A --> B |
| sig A { f: one B } | f is a total function from A to B | f : A --> B |
| sig A { f: set B } | f is a relation from A to B | f : A <-> B |
| sig Dictionary extends Book {…}  sig Novel extends Book {…} | Inheritance, all extension signatures are disjoint. | CONSTANTS  Novel, Dictionary  PROPERTIES  Dictionary ⊆ Book &  Novel ⊆ Book &  Novel ∩ Dictionary = {} |
| abstract sig Book {…}  sig Dictionary extends Book {…}  sig Novel extends Book {…} | Abstract signature, has no proper instance; all instances are obtained from extensions | PROPERTIES  …  Novel ∪ Dictionary = Book |
| one sig Bible extends Book {…} | Singleton, |Bible| = 1, Bible subset of Book | PROPERTIES  Bible ⊆ Book &  card(Bible) = 1 |
| sig LNCS in Book {…} | LNCS subset of Book. It may overlap with other extensions of Book | PROPERTIES  LNCS ⊆ Book |

**Boolean Operators**

|  |  |  |
| --- | --- | --- |
| p and q, p && q | Conjunction | p & q |
| p or q, p || q | Disjunction | p or q |
| p implies q, p => q | Implication | p => q |
| p implies e1 else e2 | Conditional expression (e1, e2 can be of any type or a formula) | B allows implication only between formulas  (p => e1) & ((not p) => e2) |
| p iff q, p <=> q | Equivalence | p <=> q |
| not p, !p | Negation | not p |

**Quantification**

|  |  |  |
| --- | --- | --- |
| all x1,…,xn : S1, …, y1,…,yn : Sm | p | Universal quantification | !(x1,…,xn,…, y1,…,yn).  (  x1 : S1 & … & xn : S1  … & … & …  y1 : S2 & … & yn : Sm  =>  p  ) |
| some x1,…,xn : S1, …, y1,…,yn : S2 | p | Existential quantification, at least one | #(x1,…,xn,…, y1,…,yn).  (  x1 : S1 & … & xn : S1  … & … & …  y1 : S2 & … & yn : Sm  &  p  ) |
| one x : S | p | Exactly one assignment of values to variables satisfies p. Also allowed for list of variables. | #(x).(x : S & p)  & !(x1,x2).  (  x1:S  & x2:S  & p[x:=x1]  & p[x:=x2]  =>  x1=x2) |
| no x : S | p | No assignment of values to variables satisfies p. Also allowed for list of variables. | not (#(x).(x : S & p)) |
| lone x : S | p | At most one assignment of values to variables satisfies p. Also allowed for list of variables. | (… one …) or (… no …) |

**Sets (ie, unary relations)**

|  |  |  |
| --- | --- | --- |
| none | The empty set | {} |
| univ | All instances of all types (the universe) | N/A |
| Int | set of integers, defined in module util/integer  The range of integers is defined by the scope  run … for *n* int  where *n* is the number of bits used to represent a signed integer. Thus, the range is -2n-1 .. (2n-1)-1.  ex: for 3 int is the interval -4 .. 3 | NAT with MININT = -2n-1 and MAXINT = (2n-1)-1 |

**Predefined Binary relations**

|  |  |  |
| --- | --- | --- |
| iden | Identity relation on univ, ie, the relation  {x:univ,y:univ | x=y} | not available  The B expression  id(S)  is the Alloy expression  S <: iden  where <: is Alloy's prerestriction operator |

**Predicates on relation**

|  |  |  |
| --- | --- | --- |
| no x | Empty set | x = {} |
| some x | Relation not empty | x /= {} |
| one x | |x| = 1 | card(x) = 1 |
| lone x | |x| <= 1 | card(x) <= 1 |
| a in B | Subset or equal | a <: B |
| a = b | Equality | a = b |
| a != b | Inequality | a /= b |
|  |  |  |

**Operators on relations**

|  |  |  |
| --- | --- | --- |
| a->b | Cartesian product a × b | a\*b |
| {x1:S,…,xn:Sn | p} | Set of tuples | {(x1,…,xn) | x1:S1 & … & xn:Sn & p}  type of set elements is ((S1\*S2)\* …)\*Sn |
| b.author | Field access. Same as set of images of b by relation author | author[{b}] |
| r1.r2 | Relation product | r1;r2 (only when r1 and r2 are binary relations)  Alloy has n-ary relations; B only has binary relations |
| a.b | Relational product extended to arbitrary *nary*-relations | N/A |
| b[a] | same as a.b | b[a]  works only if b is a binary relation and a is a set |
| x + y | Union | x \/ y |
| x & y | Intersection | x /\ y |
| x - y | Difference | x - y |
| a <: b | Domain restriction of relation b by set a | a<|b |
| b :> a | Range restriction of relation b by set a | b|>a |
| ~a | Inverse | a~ |
| \*a | Reflexive-transitive closure | closure(a) |
| ^a | Transitive closure | closure1(a) |
| a++b | Relational override,  ie, returns (a-(b.univ)) + b | a<+b |
| #a | Cardinality | card(a) |

**Types, constraints and multiplicities**

|  |  |  |
| --- | --- | --- |
| r in T->U | Relation from T to U | r in T <-> U |
| r in T -> one U | Total function from T to U | r in T --> U |
| r in T -> lone U | Partial function from T to U | r in T +-> U |
| r in T lone -> lone U | Partial injection from T to U | r in T >+> U |
| r in T lone -> one U | Total injection from T to U | r in T >-> U |
| r in T some -> lone U | Partial surjection from T to U | r in T +->> U |
| r in T some -> one U | Total surjection from T to U | r in T +->> U |
| r in T one -> lone U | Partial bijection from T to U | r in T >+>> U |
| r in T one -> one U | Bijection from T to U | r in T >->> U |

**Integers (operators defined in module util/integer)**

|  |  |  |
| --- | --- | --- |
| plus[a,b] | Sum | a+b |
| minus[a,b] | Difference | a-b |
| mul[a,b] | Product | a\*b |
| div[a,b] | Integer division | a/b |
| rem[a,b] | Remainder of a divided by b |  |
| sum[a] | Returns the sum of the integers of set a |  |
| a < b, a = b, a > b, a =< b, a >= b | Integer comparison | a < b, a = b, a > b, a <= b, a >= b |
| max[a] | Maximum of set a | max(a) |
| min[a] | Minimum of set a | max(a) |

**Global Assertions**

|  |  |  |
| --- | --- | --- |
| fact {  f1  …  f2  } | Formulas f1,…,fn which must be satisfied by all instances of a model.  Formulas f1,…,fn are implicitly conjoined. | PROPERTIES  f1 & … & fn |

**Syntactic Sugar**

|  |  |
| --- | --- |
| author[b] | b.author |
| author[Book] | Book.author |
| p1.friend[p2] | friend[p1,p2] |
| let v = E | F | Equivalent to F where v is replaced by E |

**Ordering (operators defined in module util/ordering)**

|  |  |
| --- | --- |
| open util/ordering[State] as states | Declares a total order on State |
| states/first | First element |
| states/last | Last element |
| states/next[s] | Next element |
| states/prev[s] | Previous element |
| states/nexts | All next elements |
| states/prevs | All previous elements |

**Sequences**

|  |  |
| --- | --- |
| s : seq A | Sequence |
| s.append[t] | Concatenation |
| s.first | Head |
| s.rest | Tail |
| s.elems | Unordered elements |

**Modules**

|  |  |  |
| --- | --- | --- |
| open util/ordering[States] as mystates | Opens module ordering and declares mystates as prefix for using it (ie, mystates /*function*) |  |
| module util/ordering[exactly elem] | Declares module ordering with parameter elem |  |

**Predicates and functions**

|  |  |  |
| --- | --- | --- |
| pred wrote[a:Author,b:Book]  {b.author=a} | Predicate (returns true or false) | DEFINITIONS wrote(a,b) == author[{b}] = {a} |
| fun books[a:Author]:set Book  {author.a} | Function, returns an expression of some type, here it returns a set of books |  |
| fun nbOfBooks[a:Author]:Int {#(author.a)} | Function, returns an integer. |  |

**Finding an instance of a model**

|  |  |  |
| --- | --- | --- |
| run {…} for *n* | Find instances, by default with a maximum of *n* instances for each signature (*n* is some natural number). |  |
| run {…} for 3 Book, 4 Author | Find instances with constraints on # of instances |  |
| run {…} for 3 but 1 Author | Find instances with constraints on # of instances, here 3 instances of all signatures except Author, for which only 1 instance is used. |  |
| pred foo[b:Book] {…}  run foo for 3 but 1 Author | Find instances satisfying predicate "foo" |  |

**Checking an assertion of a model**

|  |  |  |
| --- | --- | --- |
| assert assertion1  {good\_author => good\_book}  check assertion1 for … | Find counter-examples violating the assertion.  Same scope specification behavior as the run command |  |
| check nom\_check  {good\_author => good\_book} for … | Check specified assertion.  Assertion has the name nom\_check |  |
| check {good\_author => good\_book} for … | Check anonymous assertion |  |

**Precedence**

(In increasing order; operators on the same line have same priority)

|  |  |
| --- | --- |
| *Expressions (operands are not Booleans)* | *Logical expression (operands are Booleans)* |
| ~ , ^ , \*  .  []  <: , :>  ->  &  ++  #  +, -  no , some , lone , one , set  ! , not  in , = , < , > , = , =< , => | ! , not  && , and  => , implies , else  <=> , iff  || , or  let , no , some , lone , one , sum (*quantification*) |

All binary operators associate to the left, with the exception of implication, which associates to the right. So, for example, a.b.c is parsed as (a.b).c, and p => q => r is parsed as p => (q => r).