Alloy: A Quick Reference and an interpretation into B

Marc Frappier, *2020-11-09*
version 2.1

Inspired from the document Alloy Quick Reference written by *Martin Monperrus*
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.
- 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

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

Boolean Operators

<table>
<tr><td>p and q, p && q</td><td>Conjunction</td><td>p & q</td></tr>
<tr><td>p or q, p || q</td><td>Disjunction</td><td>p or q</td></tr>
<tr><td>p implies q, p => q</td><td>Implication</td><td>p => q</td></tr>
<tr><td>p implies e1 else e2</td><td>Conditional expression (e1, e2 can be of any type or a formula)</td><td>B allows implication only between formulas
(p => q1) & ((not p) => q2)</td></tr>
<tr><td>p iff q, p <=> q</td><td>Equivalence</td><td>p <=> q</td></tr>
<tr><td>not p, !p</td><td>Negation</td><td>not p</td></tr>
</table>

Quantification

<table>
<tr><td>all x1,…,xn : S1, …, y1,…,yn : Sm | p</td><td>Universal quantification</td><td>!(x1,…,xn,…, y1,…,yn).
(
  x1 : S1 & … & xn : S1
  …   & … & …
  y1 : S2 & … & yn : Sm
=>
  p
)</td></tr>
<tr><td>some x1,…,xn : S1, …, y1,…,yn : S2 | p</td><td>Existential quantification, at least one</td><td>#(x1,…,xn,…, y1,…,yn).
(
  x1 : S1 & … & xn : S1
  …   & … & …
  y1 : S2 & … & yn : Sm
  &
  p
)</td></tr>
<tr><td>one x : S | p</td><td>Exactly one assignment of values to variables satisfies p. Also allowed for list of variables.</td><td> #(x).(x : S & p)
& !(x1,x2).
    (
        x1:S
      & x2:S
      & p[x:=x1]
      & p[x:=x2]
    =>
      x1=x2)</td></tr>
</table>

<table>
<tr><td>no x : S | p</td><td>No assignment of values to variables satisfies p. Also allowed for list of variables.</td><td>not (#(x).(x : S & p))</td></tr>
<tr><td>lone x : S | p</td><td>At most one assignment of values to variables satisfies p. Also allowed for list of variables.</td><td>(… one …) or (… no …)</td></tr>
</table>

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 -2^{n-1} .. $(2^{n-1})-1$. ex: for 3 int is the interval -4 .. 3	NAT with MININT = -2^{n-1} and MAXINT = $(2^{n-1})-1$

Predefined Binary relations

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

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 max[a]	Integer comparison Maximum of set a	a < b, a = b, a > b, a <= b, a >= b 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)

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

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)`.