What is Alloy?

Lightweight modelling language for software design

Software specification using first-order predicate logic and relational calculus

Developed at MIT

Quite some number of use-cases

Fractal

other at: http://alloy.mit.edu/community/models
How is it used?

User: Specifies a meta-model and constraints (i.e. logic theory)
User: Specifies up to how many instances of each class in the meta-model to construct

Alloy analyzer: Constructs an instance of the meta-model that satisfies the constraints (i.e. it model for the given theory)
Example – *components.als*

```
sig Method { 
} { 
    some t: InterfaceType | this in t.methods 
}

sig InterfaceType { 
    methods: some Method 
} { 
    some i: Interface | i.ifaceType = this 
}

abstract sig Interface { 
    ifaceType: InterfaceType, 
} { 
    one c: Component | this in c.ifaces 
}
```
sig ProvidedInterface extends Interface {
}

sig RequiredInterface extends Interface {
    boundTo: one ProvidedInterface
}

    compatibleInterfaceTypes[ifaceType, boundTo.@ifaceType]

}

sig Component {
    ifaces: set Interface
}

pred compatibleInterfaceTypes[req: InterfaceType, prov: InterfaceType] {
    req.methods in prov.methods
}

run { } for 2
Verification / testing?

In theory this is there are an infinite number of models.

User has to specify the bounds for searching.

It turns out that if there is a problem in the theory, a small model can very often already reveal it.

The analyzer does exhaustive verification for all models that have up to a certain number of instances.

But larger models than a certain threshold are not investigated.

Thus it’s kind of mixture between verification and testing.
How does the analyzer work?

Translates the predicate logic into relational calculus

Constructs a boolean matrix for each relation

Converts the problem to a number of CNF clauses linked by logical disjunction

Runs a SAT solver for each CNF clause

Reinterprets the valuation of the boolean variables back to the language of models
Language – classes and relations

Definition of classes

sig, one sig, abstract sig, extends

sig ProvidedInterface extends Interface { }

Everything is a set

Sig declares a set of related concepts

one sig constructs a set of just one element
one sig CompA extends Component { }

Relation is a set of tuples
tuple elements are instances of classes (signatures)
Signatures

```alloy
sig A {}  
set of atoms A

sig A {}  
sig B {}  
 disjoint sets A and B (no A & B)

sig A, B {}  
same as above

sig B extends A {}  
set B is a subset of A (B in A)

sig B extends A {}  
sig C extends A {}  
B and C are disjoint subsets of A  
(B in A && C in A && no B & C)

sig B, C extends A {}  
same as above

abstract sig A {}  
sig B extends A {}  
sig C extends A {}  
A partitioned by disjoint subsets B and C  
(no B & C && A = (B + C))

sig B in A {}  
B is a subset of A – not necessarily  
 disjoint from any other set

sig C in A + B {}  
C is a subset of the union of A and B

one sig A {}  
lone sig B {}  
some sig C {}  
A is a singleton set  
B is a singleton or empty  
C is a non-empty set
```

Figure from: Greg Dennis and Rob Seater: "Alloy Analyzer 4 Tutorial"
Set operations

Standard set operators

\( e_1 + e_2, e_1 \& e_2, e_1 - e_2 \) – union, intersection, difference

\( e_1 \text{ in } e_2 \) – inclusion

Relation operators

\( e_1 \to e_2 \) – product

\( e_1 \cdot e_2, e_2[e_1] \) – join

- “inner join” on rightmost element of \( e_1 \) and leftmost element of \( e_2 \); the actual pivot columns are discarded

\( e_2 \lhd e_1 \) – domain restriction of \( e_1 \) to \( e_2 \)

\( e_1 \rhd e_2 \) – range restriction of \( e_1 \) to \( e_2 \)

\( \sim e \) – transposition

Transitive closure

\( ^e, *e \) – normal and reflexive transitive closure (works for pairs)

Standard quantifiers and cardinality predicates

all, some, one, lone, no
Relations

A set of tuples
tuple elements are instances of classes (signatures)

The relation is defined within a signature, but acts as a globally accessible set of tuples

sig InterfaceType {
    methods: some Method
}

Thus one can write (every interface type has at least one method):

- all t: InterfaceType | some m: Method | t->m in methods

or alternatively

- all t: InterfaceType | some t.methods

or even alternatively

- InterfaceType in methods.Method
Language – relations

Pre-defined sets and relations
iden – identity
univ – all elements
non – empty set
open components

fact componentNotBoundToItself { all c: Component, i: c.ifaces => RequiredInterface | not i.boundTo in c.ifaces }

fun getCompCallGraph: Component->Component { { caller, callee: Component | some i: caller.ifaces => RequiredInterface | i.boundTo in callee.ifaces } }

pred cycleInArchitecture { some iden & ^getCompCallGraph }

check { !cycleInArchitecture } for 4
Unsat core

Alloy is able to help finding contradicting clauses in overconstrained model

Example – *unsat core.als*
Dynamic systems in Alloy

The theory may also contain description of systems dynamics

Theory describes a valid trace

The model contains a particular trace

Example – farmer.als
What next? – Our possible research

Component reconfiguration models specified and tested in Alloy

Potentially connector (or even micro-component) architecture resolver using SAT

Either Alloy directly

or Kodkod – underlying Java library allowing for

- transformation of relational calculus to SAT with some optimizations
- calling an existing SAT solver

Support for multistage theories
What next? – Support for multistage theories

Theory: Component architecture

Meta-model: Component arch.

Meta-model: Modes

Meta-model: Component arch.

Meta-model: Mode switch. trace

Meta-model: Modes

Meta-model: Component arch.

Instance: Component architecture

Theory: Modes

Instance: Modes

Theory: Mode switching

Instance: Component architecture

Meta-model: Modes
Q & a