Alloy and the Alloy Analyzer

Klaus Reisenberger

JKU Linz

Klaus.Reisenberger@gmx.at

November 23, 2014

Outline

1. Introduction
2. The Language
3. Example
4. Alloy Analyzer
5. Conclusion

Section 1

Introduction

- Alloy: A language for describing structures
- Alloy Analyzer: A tool for exploring this structures.

History

- Developed by the Software Design Group at MIT lead by Professor Daniel Jackson in 1997.
- First success with VDM (Vienna Development Method) in 1980 .
- During his PhD at MIT he got intrigued by Larch (system for formal specification and verification of program modules).
- Theorem proving could not fully be automated and formal models where hard to construct.
- In 1992 became fond of the Z language (less complex than previous languages, based on the simplest notions of set theory).
- But it was even less analyzable than the Z language.
- Idea of Alloy: bring the power of modelcheckers to Z.

Alloy

- http://alloy.mit.edu/alloy
- Deeply rootet in Z.
- It has a simpler language.
- Analysis relies on SAT (boolean satisfiability).

Alloy Analyzer

- Freely available online for a variety of platforms
  http://alloy.mit.edu/alloy/download.html
- Translates constraints written in Alloy into boolean constraints and passes them to a SAT solver.

Section 2

The Language

Signatures

A Signature is a set of atoms.

Syntax

- **sig** A { }
- **sig** A, B { }
- **abstract sig** A { }
  sig B **extends** A { }
  sig C **extends** A { }

Example

```
abstract sig Person { }

sig Woman extends Person { }

sig Man extends Person { }
```

Person is partitioned by the disjoint subsets Woman and Man.

Multiplicity

- set: zero or more
- one: exactly one
- lone: zero or one
- some: one or more

Fields

After the signature comes its body. It defines relations with its signature as domain.

Syntax

- **sig** A {f : e }
  So f is a binary relation with domain A and range e.

Example

```
abstract sig Person {
father: one Man,
mother: one Woman
}

sig Woman extends Person { }

sig Man extends Person { }
```

Every person has exactly one father and mother.

Facts

Facts are constraints that are assumed always to hold. They restrict the model.

Syntax

- **fact** { F }
- **sig** A { . . . } { F }

Example

```
sig Phone { }

sig Call {from, to: Phone} { from != to }

fact { all x: Call | x.from != x.to }
```

Functions

Functions are named expression intended for reuse.

Syntax

- **fun** f $[x1 : e1, \ldots, xn : en]$: e { E }

Example

```
fun grandpas[p: Person] : set Person {
p.(mother + father).father
}
```

So the grandpas of a person are the fathers of one's own mother and father.

Predicates

Predicates are constraints that you don't want to record as facts. (e.g.,you might want to analyze a model with a particular constraint included, and then excluded)

Syntax

- **pred** $p\,[x1 : e1, \ldots, xn : en]\ \{\ E\ \}$

Example

```
pred ownGrandpa [p: Person] {
p in grandpas[p]
}
```

Assertions

Assertions are constraints that are expected to follow from the facts of the model. The analyzer checks assertions to detect design flaws.

Syntax

- **assert** *a* { *F* }

Example

```
sig Node {
children: set Node
}

one sig Root extends Node {}

fact {
Node in Root.*children
}

assert someParent {
all n: Node - Root |  some children.n
}
```

Check

The check command instructs the analyzer to search for counterexample to assertions within scope.

Syntax

check *a scope*

Example

```
fact {
no p: Person | p in p.^(mother + father)
wife = ~husband
}
assert noSelfFather {
no m: Man | m = m.father
}
check noSelfFather
```

So we search for a counterexample to noSelfFather within a scope of at most 3 Persons (default)

Operators

- Set operators
  - Union $+$
  - Intersection &
  - Difference –
  - Subset **in**
  - Equality $=$
- Product operator: – >
- Operators on relations
  - Transpose: $\sim$
  - Transitive closure:ˆ
  - Reflexive transitive closure: *
- Boolean operators
  - Negation: ! not
  - Conjunction: && and
  - Disjunction: || or
  - Implication: $=>$ implies
  - Alternative: else
  - Bi-implication: $<=>$ iff

Quantifiers

Syntax

- all x:e | F
- all, some, one, lone, no

Section 3

Example

File system - Part 1

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }

// A directory is the parent of its contents
fact { all d: Dir, o: d.contents | o.parent = d }

// All file system objects are either files or directories
fact { File + Dir = FSObject }

// There exists a root
one sig Root extends Dir { } { no parent }

// File system is connected
fact { FSObject in Root.*contents }
```

File system - Part 2

```
// The contents path is acyclic
assert acyclic { no d: Dir | d in d.^contents }

// Now check it for a scope of 5
check acyclic for 5

// File system has one root
assert oneRoot { one d: Dir | no d.parent }

// Now check it for a scope of 5
check oneRoot for 5

// Every fs object is in at most one directory
assert oneLocation { all o: FSObject | lone d: Dir | o in d.contents }

// Now check it for a scope of 5
check oneLocation for 5
```

Section 4

Alloy Analyzer

C:\Users\Klaus\Desktop\alloy analyzer.als

File Edit Execute Options Window Help

New Open Reload Save Execute Show

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }

// A directory is the parent of its contents
fact { all d: Dir, o: d.contents | o.parent = d }

// All file system objects are either files or directories
fact { File + Dir = FSObject }

// There exists a root
one sig Root extends Dir { } { no parent }

// File system is connected
fact { FSObject in Root.*contents }

// The contents path is acyclic
assert acyclic { no d: Dir | d in d.^contents }

// Now check it for a scope of 5
check acyclic for 5

// File system has one root
assert oneRoot { one d: Dir | no d.parent }

// Now check it for a scope of 5
check oneRoot for 5

// Every fs object is in at most one directory
assert oneLocation { all o: FSObject | lone d: Dir | o in d.contents }

// Now check it for a scope of 5
check oneLocation for 5
```

Line 2, Column 34 [modified]

```
Executing "Check oneRoot for 5"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1016 vars. 62 primary vars. 1470 clauses. 46ms.
   No counterexample found. Assertion may be valid. 12ms.

Executing "Check oneLocation for 5"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1079 vars. 67 primary vars. 1543 clauses. 42ms.
   No counterexample found. Assertion may be valid. 2ms.

Executing "Check acyclic for 5"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1072 vars. 67 primary vars. 1803 clauses. 35ms.
   No counterexample found. Assertion may be valid. 25ms.

Executing "Check oneRoot for 5"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1016 vars. 62 primary vars. 1470 clauses. 45ms.
   No counterexample found. Assertion may be valid. 25ms.

Executing "Check oneLocation for 5"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1079 vars. 67 primary vars. 1543 clauses. 65ms.
   No counterexample found. Assertion may be valid. 2ms.

3 commands were executed. The results are:
   #1: No counterexample found. acyclic may be valid.
   #2: No counterexample found. oneRoot may be valid.
   #3: No counterexample found. oneLocation may be valid.

Executing "Check oneLocation for 5"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1079 vars. 67 primary vars. 1543 clauses. 41ms.
   No counterexample found. Assertion may be valid. 2ms.

Metamodel successfully generated.
```

Section 5

Conclusion

Applications

Alloy is used in many applications:
http://alloy.mit.edu/alloy/applications.html

- Equals Checker: A tool for checking equal methods in Java.
- Nitpick: A counterexample generator for Isabelle/HOL.
- Margrave: A security policy analyzer for firewalls.
- Secrecy Modeling Language: A language for composing and validating security models.

References

- Jackson, Daniel:
  Software Abstractions : Logic, Language, and Analysis.
  Cambridge: MIT Press, 2012.
- Alloy MIT Online Tutorial
  Retrieved November 19, 2014, from
  http://alloy.mit.edu/alloy/tutorials/online/
- Edward Yue Shung Wong, Michael Herrmann, Omar Tayeb
  A Guide To Alloy
- Alloy 4 Tutorial Materials
  Retrieved November 19, 2014, from
  http://alloy.mit.edu/alloy/tutorials/day-course/