Lecture on Automated Software Engineering

Alloy: Analyzing Software Requirements, Design and Algorithms

Martin Monperrus, Ph.D.

version of Oct 15, 2012

The latest version of these slides can be found at: http://www.monperrus.net/martin/alloy-slides.pdf

Content of this lecture

These slides

- provide an overview of the language "Alloy"
- provide an overview of the different usage of Alloy:
  - generate instances (e.g. test cases)
  - detect overspecifications
  - detect underspecifications
  - verify properties

- See also "Alloy: A Quick Reference" http://www.monperrus.net/martin/alloy-quick-ref.pdf

Content of this lecture

These slides

- provide an overview of the language "Alloy"
- provide an overview of the different usage of Alloy:
  - generate instances (e.g. test cases)
  - detect overspecifications
  - detect underspecifications
  - verify properties

- See also "Alloy: A Quick Reference"
  http://www.monperrus.net/martin/alloy-quick-ref.pdf

http://alloy.mit.edu

Dependable Software by Design

Computers fly our airliners and run most of the world's banking, communications, retail and manufacturing systems. Now powerful analysis tools will at last help software engineers ensure the reliability of their designs

By Daniel Jackson

References:

- **ALCOA: The Alloy constraint analyzer, 2000**
- **Automating First-Order Relational Logic, 2000**
- **Finding bugs with a constraint solver, 2000**
- **A micromodularity mechanism, 2001**
- **Alloy: a lightweight object modelling notation, 2002**
- **Software Abstractions Logic, Language, and Analysis, 2006**

What is Alloy?

Alloy is a tool to find conceptual bugs (~~not implementation bugs~~): domain logic, communication protocols, emergent properties, etc.

A bug is a property which is not verified:

- No explicit property = no bug found

"No planes can be allowed to land at the same time"

The intuition

The core idea of Alloy is transform a property and the corresponding model into a first order logic formula:

$$all\ y{:}Y \mid !\, x.r = y$$

$$\neg(((x_0 \wedge r_{00}) \vee (x_1 \wedge r_{10})) \wedge \neg((x_0 \wedge r_{01}) \vee (x_1 \wedge r_{11}))) \wedge$$
$$\neg(\neg((x_0 \wedge r_{00}) \vee (x_1 \wedge r_{10})) \wedge ((x_0 \wedge r_{01}) \vee (x_1 \wedge r_{11})))$$

and to verify this model with a standard SAT solver:

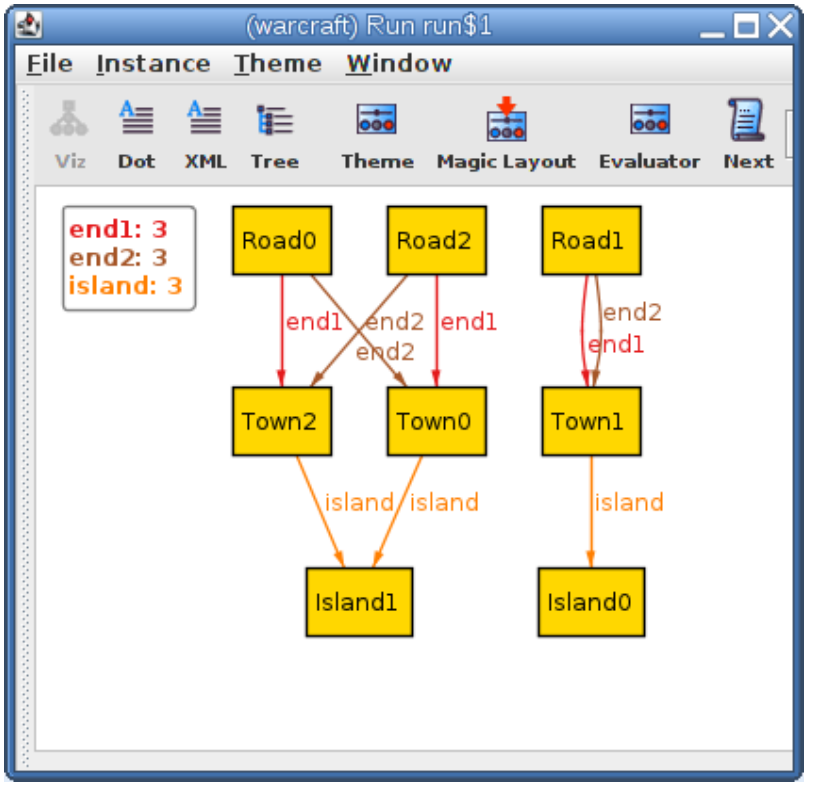
```
The formula is satisfiable:

x0=true

x1=false

r00=true

etc.
```

Generating instances with Alloy

Alloy consists of generating instances of an object-oriented model.

Your first Alloy Program: a WoW Map Generator

```
sig Island {  }
sig Town { island : Island }
sig Road {
  end1 : Town,
  end2 : Town
}

fact {
no i,j: Island | some r : Road |
  i!=j and r.end1.island =i and r.end2.island =j
}

run { } for 3
```



A basic Alloy model consists of signatures and facts.

Detecting overspecification with Alloy (no instances)

Overspecification: assumption

In case of overspecification, there are no possible instances. This appears even in presence of slight overspecification.

A natural language specification (1)

- A file system object has a parent
- A directory is a special kind of file system object
- A directory contains file system objects
- There is one directory which is called the root
- The root directory has no parent

```
// A file system object has a directory as parent
sig FSObject {
  parent: Dir
}

// A directory is a special kind of file system object
sig Dir extends FSObject {
// A directory contains file system objects
  contents:  set FSObject
}

// There is one directory which is called the root
one sig Root extends Dir {}

// The root directory has no parent
fact RootProperty { no Root.parent }

run {} for 5
```

Overspecifications are detected with the absence of instances.

- Overspecification are detected by "No Instance Found"

```
Executing "Run run$1"
   Solver=sat4j Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   246 vars. 24 primary vars. 334 clauses. 46ms.
   No instance found. Predicate may be inconsistent. 67ms.
```

```
// A file system object has a directory as parent
sig FSObject {
  parent : Dir
}

// A directory is a special kind of file system object
sig Dir extends FSObject {
// A directory contains file system objects
  contents:  set FSObject
}

// There is one directory which is called the root
one sig Root extends Dir {}

// The root directory has no parent
fact { no Root.parent }

run {

}
```

~ Java class

~ UML 1..1

~ Java extends

~ UML 0..*

singleton

fact = always true

find instances

The Fix

```
// R1: All file system objects but Root have a directory as parent
sig FSObject {
  parent: lone Dir
}

// A directory is a special kind of file system object
sig Dir extends FSObject {
// A directory contains file system objects
  contents:  set FSObject
}

// There is one directory which is called the root
one sig Root extends Dir {}

// The root directory has no parent
fact RootProperty {
no Root.parent
// see R1
all t:FSObject | t not in Root implies one t.parent
}

run {}
```

~ UML 0..1

Predicate logic

A natural language specification (2)

- A file system object can have a parent
- A directory is a special kind of file system object
- A directory contains file system objects
- A file is a special kind of file system objects
- There is one special directory which is called the root and has no parent
- A directory is the parent of its contents
- Every file system object is in one directory
- A directory can not be in itself
- A directory can not be one of its ancestors
- It is possible to have directories containing several objects
- All file system objects must have one parent

Where is the bug?

Detecting Underspecifications with Alloy (wrong instances)

Underspecification: assumption

In case of underspecification, wrong instances appear quickly. The search strategies of Alloy further fasten their occurrences.

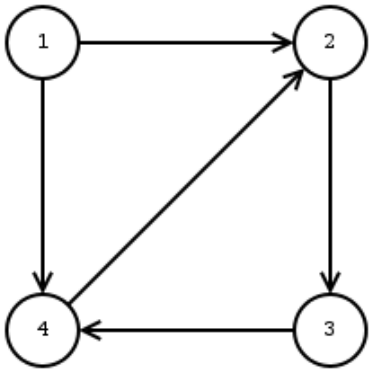
Example 1

Specification:

- A file system object is in a directory
- A directory contains file system objects
- There is no cycle in the structure

Question: is it possible?

```
sig FSObject { parent: Dir }
sig Dir { contents: set FSObject }
pred noCycle { all d:Dir | d not in d.^parent }
// question
run { some  FSObject and noCycle }
```



A transitive closure is the set of all reachable nodes.

Warning
The join operation here always yields an empty set.
Left type = {this/Dir}
Right type = {this/FSObject->this/Dir}

Alloy is a typed language, some bugs are caught at compile time.
Bug: instances of Dir have no field "parent"
Solution: add "extends Dir"

Example 1

Specification:

- A file system object is in a directory
- A directory is a file system object and contains file system objects
- There is no cycle in in the structure

Question: is it possible?

```
sig FSObject { parent: Dir }
sig Dir extends FSObject { contents: set FSObject }
pred noCycle { all d:Dir | d not in d.^parent }
// question
run { some  FSObject and noCycle }
```

No instance found!

Bug: no concept of Root directory

Solution: add the missing concept and the associated facts

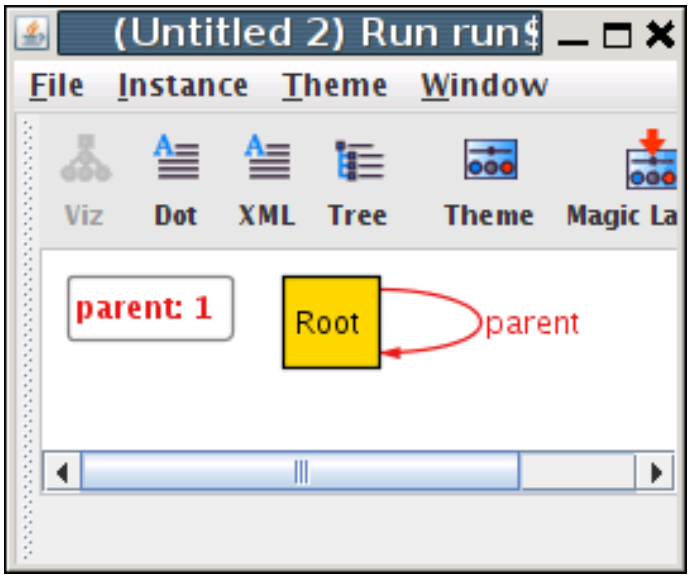
Example 1

Specification:

- A file system object is in a directory
- A directory is a file system object and contains file system objects
- There is no cycle in in the structure
- There is on directory called Root which has no parent.

Question: is it possible?

```
sig FSObject { parent: Dir }
sig Dir extends FSObject { contents: set FSObject }
pred noCycle {
all d:Dir - Root | d not in d.^parent
}
one sig Root extends Dir {}

run { some  FSObject and noCycle }
```



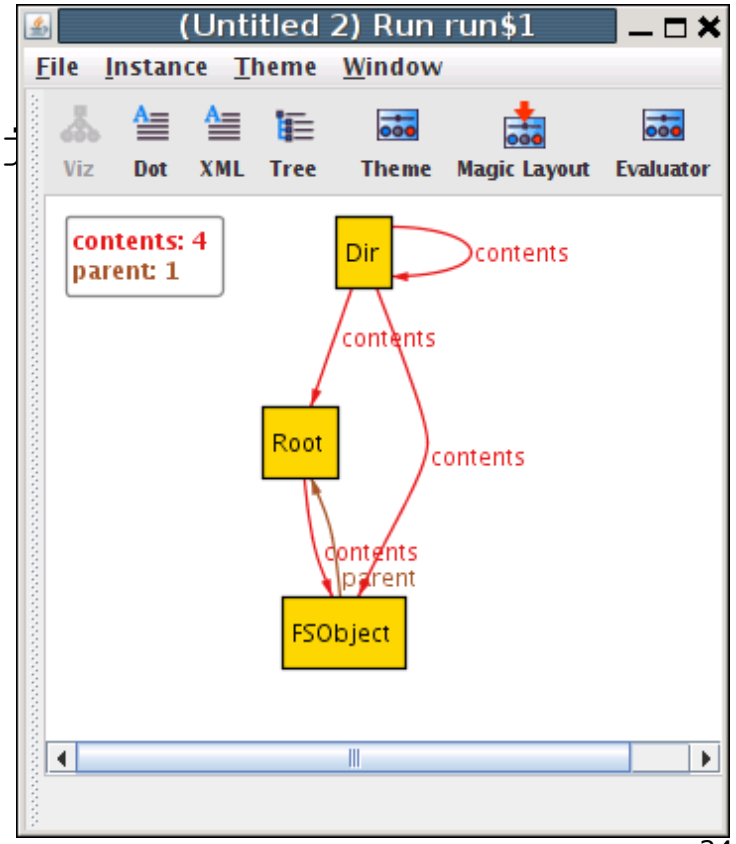
Example 1

Specification:

- A file system object is in a directory
- A directory is a file system object and contains file system objects
- There is no cycle in in the structure
- There is on directory called Root which has no parent.

Question: it it enough?

```
sig FSObject { parent: lone Dir }

sig Dir extends FSObject { contents: set FSObj

pred noCycle {

all d:Dir - Root | d not in d.^parent

}

one sig Root extends Dir {}

fact {no Root.parent}


run { some  FSObject and noCycle }
```



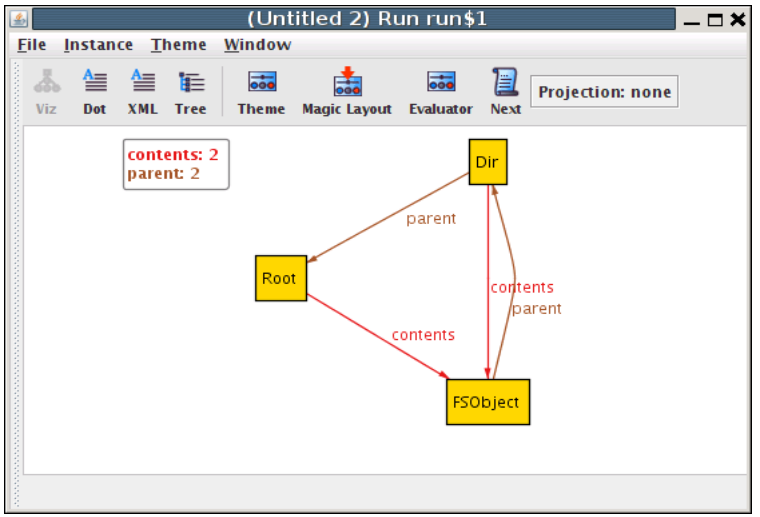
Example 1

Specification:

- A file system object is in a directory
- A directory is a file system object and contains file system objects
- There is no cycle in in the structure
- There is on directory called Root which has no parent.
- A directory is the parent of its content

Question: it it enough?

```
fact {  all d:Dir - Root | d not in d.^parent }

fact { all d:Dir |  all c:d.contents | d = c.parent }
```

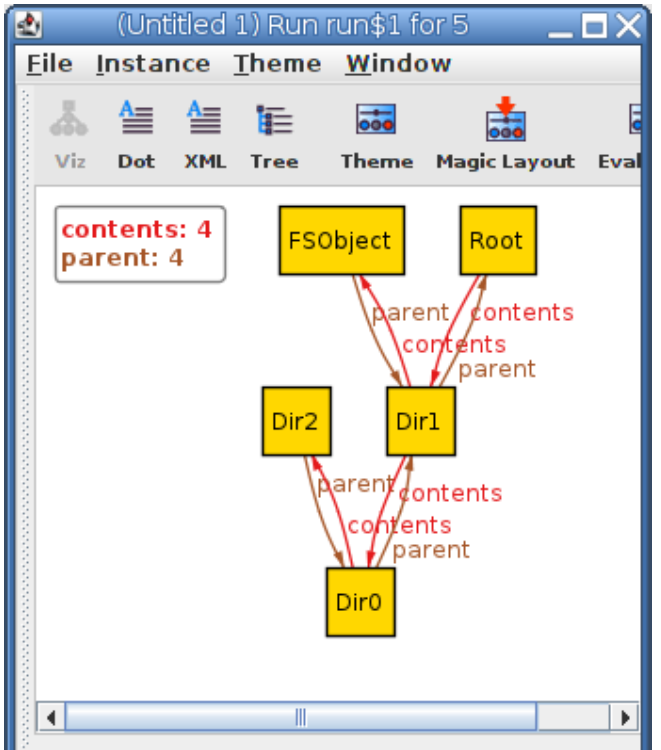


Example 1

- A file system object is in a directory
- A directory is a file system object and contains file system objects
- There is no cycle in in the structure
- There is on directory called Root which has no parent.
- A directory is the parent of its content
- All files but root are in the contents of its parent directory

Question: it it enough? YES (in a certain scope, by checking some instances)

```
fact {  all f:FSObject - Root | f  in
f.parent.contents }
```



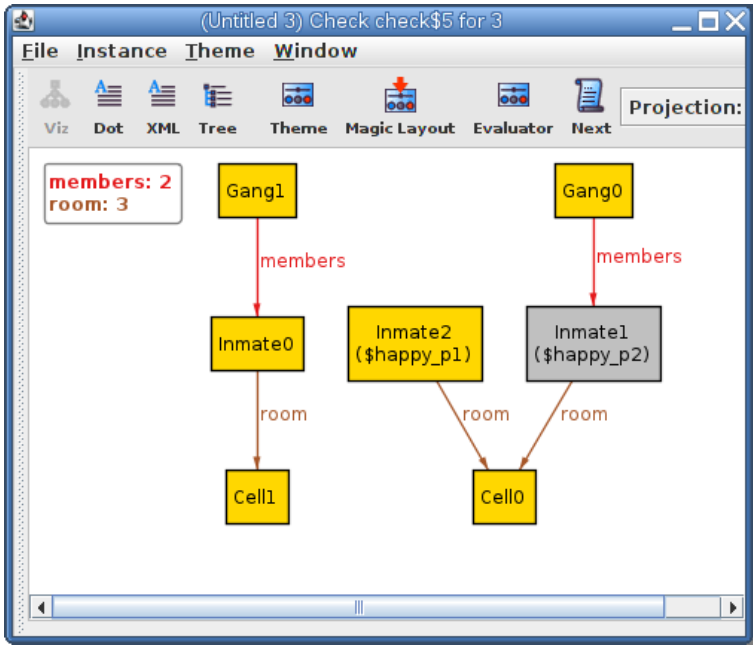
The Final Specification

```
abstract sig FSObject { parent: lone Dir }

sig Dir extends FSObject { contents: set FSObject }

sig File extends FSObject {}

one sig Root extends Dir {}

fact {no Root.parent}

fact {  all d:FSObject - Root | d not in d.^parent }

fact { all d:Dir |  all c:d.contents | d = c.parent }

fact {  all f:FSObject - Root | f  in f.parent.contents }

run { } for 6
```

Verifying Properties with Assertions

safe

```
sig Gang { members : set Inmate }
sig Inmate { room: Cell }
sig Cell { }

// no room shared
pred safe {
  no g1,g2: Gang | g1!=g2 and some (g1.members.room &
g2.members.room)
}

pred happy {
  all p1,p2 : Inmate |
    // if they are in the same room
    // they are in the same gang
    p1.room = p2.room
implies ~members[p1] = ~members[p2]
}

check {safe implies happy } for 3
```



Assertions

- An assertion derives/emerges from the rest of the world
  - e.g. N predicates implies 1 predicate
  - The N predicates will be in the implementation, not the property itself
- Two ways of expressing assertions:
  - check {safe implies happy } for 3 // anonymous
  - named assertion:

```
assert safe_implies_happy {safe implies happy }
check safe_implies_happy for 3
```

A predicate is a conditional fact.

It is used for searching for particular instances, and for writing assertions.

A predicate may have arguments.

Predicates

```
// predicate with argument
pred happy[p1 : Inmate] {
   some p2 : Inmate - p1|
    // if they are in the same room
    // they are in the same gang
    p1.room = p2.room
       and  ~members[p1] = ~members[p2]
}

// predicate used for searching for instances
run happy

// predicate used in assertions
check {safe implies happy } for 3
```

A function is a side-effect free helper.

It is used for reusing common code.

Functions

```
fun gangOf[p1 : Inmate] : Gang {
   ~members[p1]
}

pred happy[p1 : Inmate] {
  some p2 : Inmate - p1|
    // if they are in the same room
    // they are in the same gang
    p1.room = p2.room
      and  gangOf[p1] = gangOf[p2]
}
```

Summary

Alloy usages are:

- generating instances
- detecting overspecifications
- detecting underspecifications
- verifying properties with assertions

In order to:

- analyze specifications (e.g. of file systems)
- analyze algorithms (e.g. linked list manipulation)
- solve problems in a declarative manner (e.g. Hanoi's tower)