Lecture

Alloy: Verifying System Requirements and Models

Martin Monperrus, Ph.D.

Creative Commons Attribution License
Copying and modifying are authorized as long as proper credit is given to the author.

Googlism

Google: {“alloy is” site:mit.edu}

- Alloy is a textual language developed at MIT by Daniel Jackson and his team
- Alloy is a pure ASCII notation
- Alloy is a fully declarative language
- Alloy is one of several emerging 'lightweight formal methods'
- Alloy is a lightweight modeling language
- Alloy is a constraint solver that provides fully automatic simulation and checking
- Alloy is similar to OCL, the Object Language of UML, but it has a more conventional syntax and a simpler semantics
- Alloy is relational: its underlying data structures are sets and relations.
- Alloy is an attractive teaching tool

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

COPYRIGHT 2006 SCIENTIFIC AMERICAN, INC.

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**

Alloy is a tool to find bugs:

- ~~not implementation bugs~~
- conceptual bugs: domain models, communication protocols, domain logic, 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 modelinto 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.
```

Part 1: Detecting overspecification with Alloy

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
- There root directory has no parent

Overspecifications are detected with the absence of instances.

(see 01-overspecification.als)

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 in a special kind of file system objects
- There is one directory which is called the root
- There root directory has no parent
- A directory is the parent of its contents
- 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
- Every file system object is in at most one directory

- Overspecification are detected by "No Instance Found"
  - see 01-overspecification.als and 02-overspecification.als

```
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.
```

- This is one of the core *assumption* of Alloy.
- Instance spaces are very sensitive to wrong specifications
- My (humble) own experience confirms it.

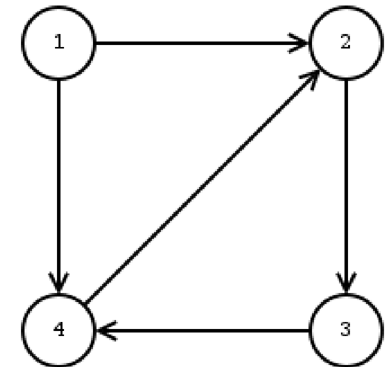
Part 2: Incremental Specification with Alloy

Example 1

Specification:

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

Question: is it possible?



A transitive closure is the set of all reachable nodes.

```
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 }
```

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

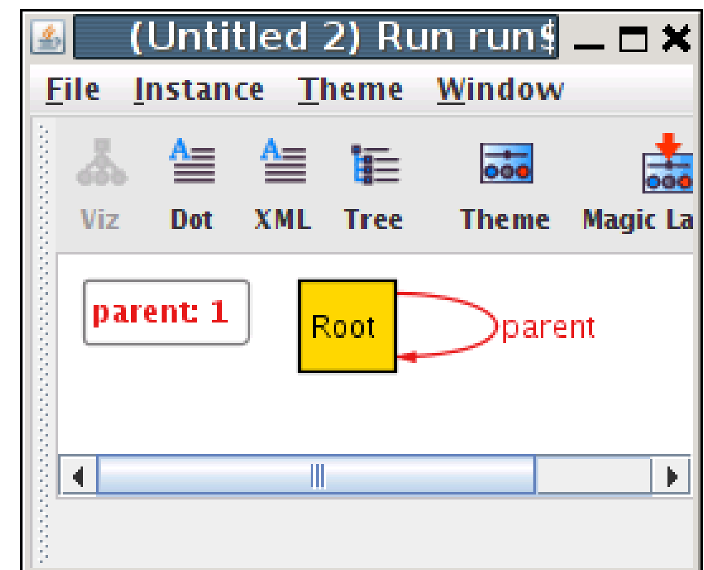
Example 1

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 }
```



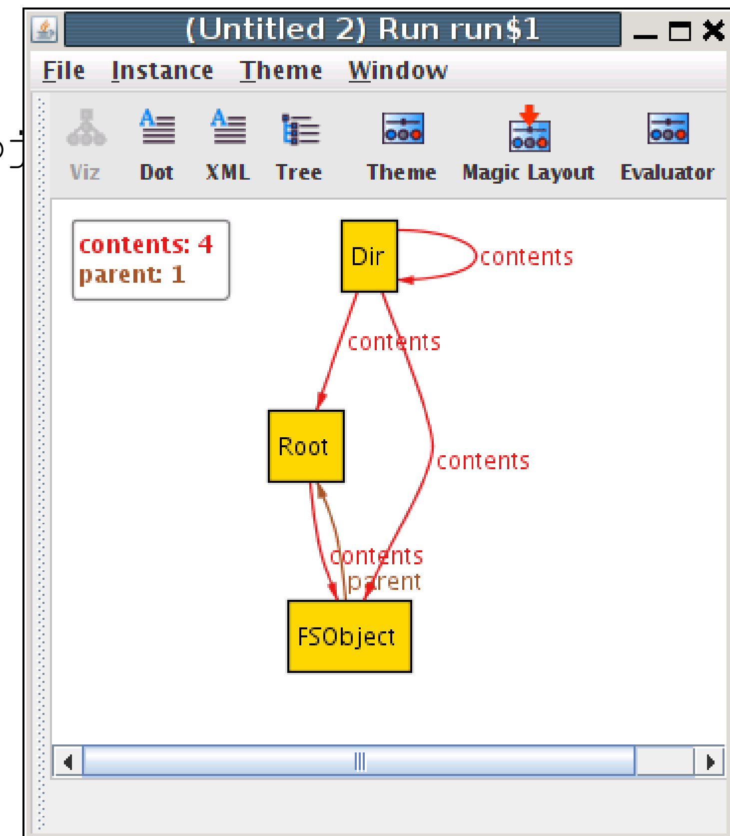
Example 1

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 }
```



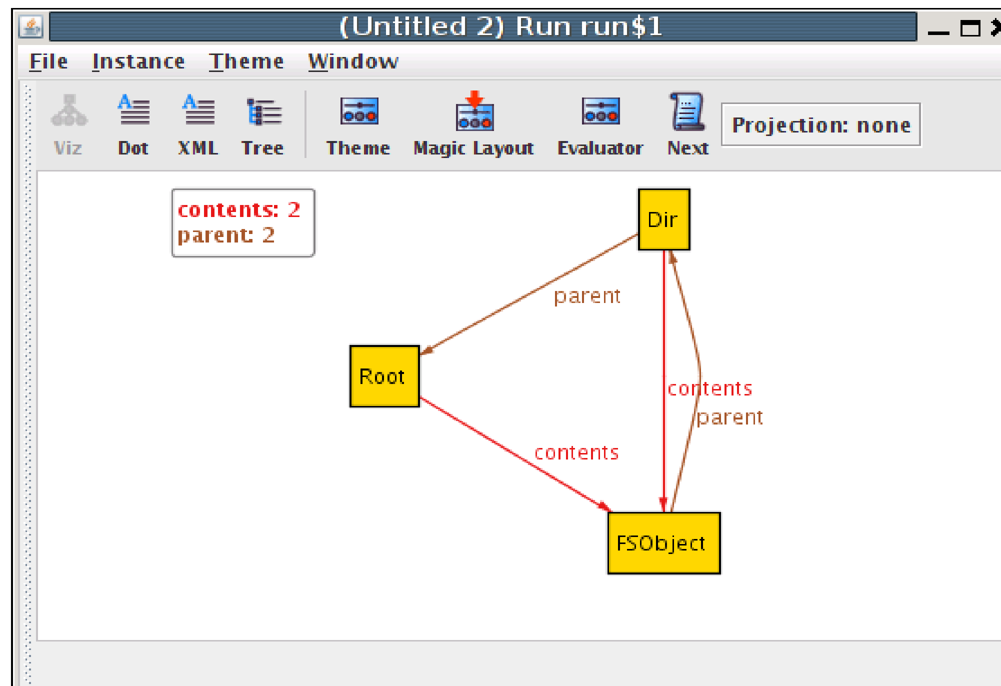
Example 1

Specification:

- A directory is the parent of its content

Question: it it enough?

```
pred noCycle {  all d:Dir - Root | d not in d.^parent and
parentOfContent[d]}

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



The Final Specification (03-incremental-specification.als)

```
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}
pred noCycle {  all d:Dir - Root | d not in d.^parent }
pred parentOfContent{
all d:Dir |  all c:d.contents | d = c.parent }
pred contentOfParent{
all f:FSObject-Root | f in f.parent.contents }
run { some  FSObject and noCycle
and parentOfContent and contentOfParent} for 6
```

Incremental Specification

Alloy helps engineers to find missing points in their specifications.

Part 3: Property verification with Alloy

Train Safety: part 1

We express the basic model.

```
sig Train {}

// the system is composed of segments

sig Seg {

// this is a directed graph

next: set Seg,


// some segments are mutually exclusive is they contain trains

// An overlap in railway signalling is the length of track

    overlaps: set Seg

}

sig SystemState {

  on: Train -> one Seg,

  occupied: set Seg

}
```

Train Safety: part 2

We now express the safety condition

```
pred Safe [x: SystemState] {
// there could not be two trains on the same segment
no t1, t2 : Train | t1 != t2 and  x.on[t1]= x.on[t2]
// this tests the overlap condition
all s: Seg | lone s.overlaps.~(x.on)
}

run p2_findnotsafe {one s:SystemState | not Safe[s]}  for 2 but 0 GateState, 1 SystemState

assert p2_NoOverlappingTrains {
no ts:SystemState, t1,t2:Train| Safe[ts] and t1!=t2 and  t1!=t2 and ts.on[t1] in ts.on[t2].overlaps
}

check p2_NoOverlappingTrains for 6 but 0 GateState, 1 SystemState
```

Train Safety: part 3

We express the notion of GateState

```
// A Gate closes a segment, hence the state of all gates

// can be reduced to the state of closed ones

sig GateState {closed: Seg }




// are two SystemStates compatible with a gatestate?

pred MayMove [g: GateState, x,x': SystemState] {

  // it is impossible to have a train on closed segments

  no g.closed.~(x'.on)

}


pred IsAfter [before,after: SystemState] {

  // all trains are in now in a "next" segment of the system

  all t: Train | t.(after.on) in t.(before.on).next

}
```

Train Safety: part 4

We express the the notion of GatePolicy

```
// we want to determine a gate policy from a given state

pred GatePolicy [g: GateState, x: SystemState] {

  all s1, s2 : Seg |

  s1 != s2 and s1 in x.occupied and s2 in x.occupied and  some s1.next.^overlaps & s2.next.^overlaps

   =>  s1.next in g.closed or s2.next in g.closed

}



// can we find a problem now?

pred p4_findAProblem[g: GateState, x,x': SystemState] {

IsAfter[x,x'] MayMove[g,x,x'] Safe[x] GatePolicy[g,x]

not Safe[x']

}

run p4_findAProblem for 5 but 2 SystemState, 1 GateState
```

Train Safety: part 4

```
assert p5_GatePolicyIsOK {

all x,x':SystemState,g:GateState | Safe[x] and GatePolicy[g,x] and IsAfter[x,x'] and
MayMove[g,x,x'] => Safe[x']

}



check p5_GatePolicyIsOK for 7 but  2 SystemState, 1 GateState
```

The specification of the gate policy is verified (up to a certain scope)

Part 4: Case Study: analysis of a multicast protocol

Source: A lightweight formal analysis of a multicast key management protocol, FORTE'2003

Secure Multicast

- Secure multicast consists of sending an identical message to a group of authorized agents (teleconferencing, distributed game, etc.).
- Messages should not be readable by unauthorized users, while it must be guaranteed that all authorized users are able to read it.
  - Usage of a group key
- So after each join or leave, a new group key is generated and distributed to all current group members in order to send or receive the messages securely.
- There is a big overhead caused by the synchronization of all members during rekeying in a highly dynamic group

Asynchronous rekeying

- Asynchronous rekeying consists of distributing the key on demand, just prior to use.
- Each domain has a trusted Key Distribution Server (KDS) which has information about its domain membership and is responsible for processing the requests of the domain's members (authentication, leave, etc).
- When a member decides to send a message, it sends a sequence number request to the KDS of its domain to check the newness of the group key it owns.
- When a member receives a message which is not encrypted by any of its valid keys, it sends a request to the KDS of its domain and asks for the newer keys.

Basic components of the model

```
open util/ordering[Tick] as tickOrd // expresses a total order over ticks
sig Tick {}
sig Member {
  kds : KDS,
  ownedKeys : Tick -> Key,
  receivedMessages : Tick -> Message }
sig KDS {
  keys : Tick -> Key,  // the keys known at this time
  members : Tick -> Member }
sig Message {
  sender : Member,
  sentTime : Tick,
  encryptingKey : Key }
sig Key { creator : KDS }
```

The failed property

```
assert OutsiderCantSend {
  no msg : Message, m : Member, t : Tick {
    !IsMember(msg.sender, msg.sentTime)
    IsMember(m, t)
    CanReceive(m, msg, t) } }
```

Time, KDS Newest Key	Member m_0, Newest Key	Member m_1, Newest Key
T_1, k_3	join, k_3	-
T_2, k_4	- , k_3	join, k_4
T_3, k_5	- , k_3	leave, k_4
T_4, k_5	- , k_3	send a message , k_4
T_5, k_5	receive the message, k_5	- , k_4

"If the member receives the message that all keys of the member cannot decrypt, the member must request the new key to the key distribution server."

```
fun NewerKeys(c : Client, kds : KDS,  t : Tick,  lastKey : GroupKey) : set GroupKey {
//  If at the given time the client is not present in the domain corresponding to the
KDS, the KDS does not send any keys.
c !in kds.members[t] => no result,
// it sends all the keys valid newer than the client’s key.
  result = kds.keys[t] & OrdNexts(lastKey)
}


fun NewerKeys2(c : Client, kds : KDS, t : Tick,
                              lastKey : GroupKey) : option GroupKey {
  c !in kds.members[t] => no result,
// it sends only the newest
  result = NewestKey(kds.keys[t] & OrdNexts(lastKey))
}
```

Conclusion

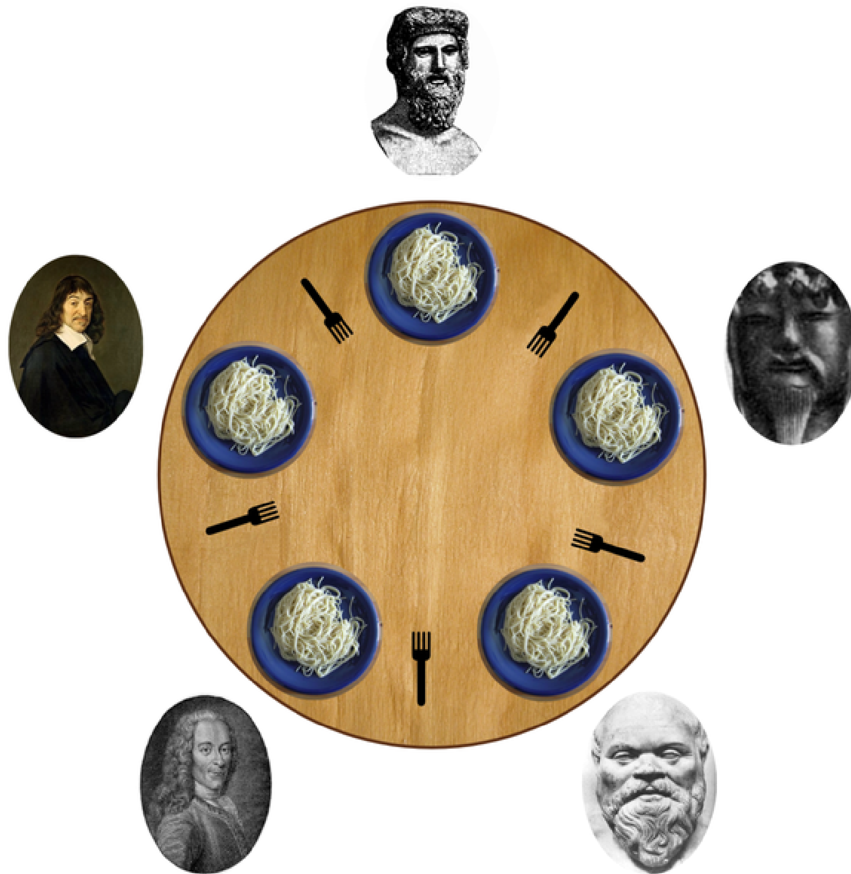
To design reliable software, Alloy helps engineers to:

- detect incorrect and over- specifications
- to precise specifications with correct instances
- to verify properties of models

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.

Executing "Check p2_NoOverlappingTrains for 6 but 0 GateState, 1 SystemState"
Solver=sat4j Bitwidth=4 MaxSeq=6 SkolemDepth=1 Symmetry=20
2932 vars. 140 primary vars. 5272 clauses. 44ms.
Counterexample found. **Assertion** is invalid. 100ms.

Exercise: dining philosophers problem



- Understand Alloy code
- Write Alloy code
- Show the presence of deadlocks
- Demonstrate the effectiveness of the "Waiter" solution