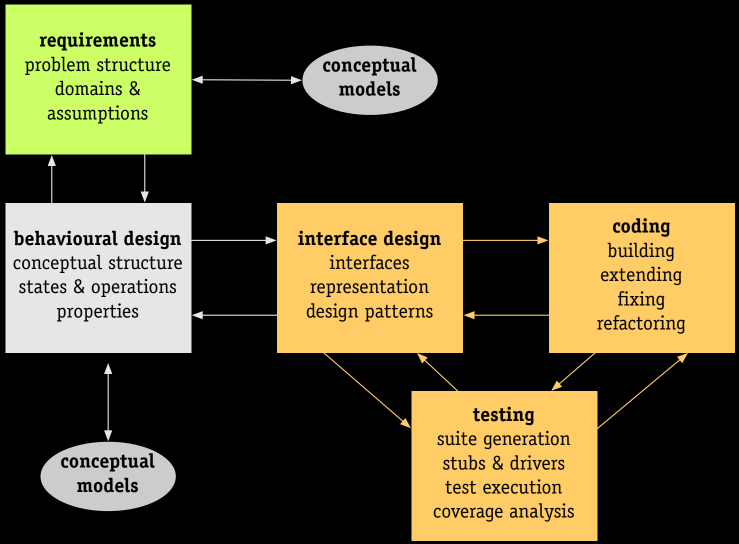
Conceptual modeling of entities and relationships using Alloy

K. V. Raghavan

Indian Institute of Science, Bangalore

two kinds of design



Conceptual modeling

What is it?

- Capture requirements, other essential aspects of software
- Abstract out inessential details
- Analyze model
  - Identify high-level errors at analysis time itself, rather than after coding

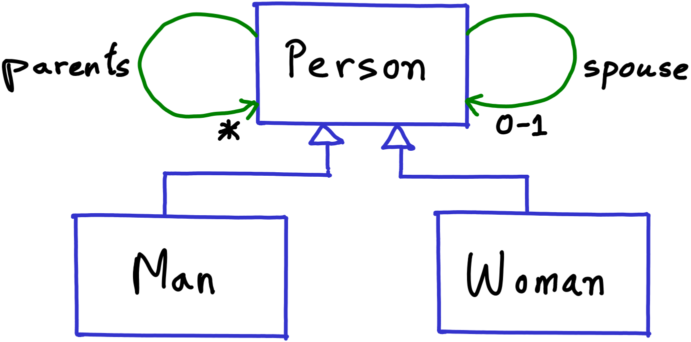
Kinds of conceptual models

- Class diagrams
- State machines
- Logics (propositional, predicate, Hoare)
- Algebras (Relational, real numbers, etc.)

Overview of Alloy

- Formal modeling of entities and associations, using sets and relations
- Modeling of invariants/constraints on the relationships
- Analyzing the model, and identifying whether it is under-constrained (i.e., allows erroneous relationships), or over-constrained (i.e., disallows required relationships)

Example – keeping track of family relationships



Examples of desired constraints

- Every person has two parents, one man and one woman,
- parents of any child are married,
- cannot marry a sibling or a parent,
- ...

Key elements of Alloy model: classes and relations

`abstract sig Person {spouse: lone Person, parents: set Person}`

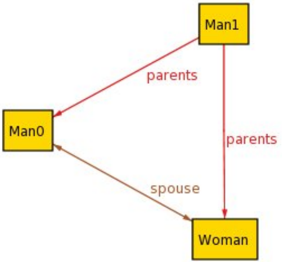
- `Person` is a *abstract* class (i.e., with no concrete objects).
- `spouse` is a relation mapping each `Person` to zero or one `Person`
- `parents` is a relation mapping each `Person` to zero or more `Persons`

`sig Man, Woman extends Person {}`

- `Man, Woman` are subtypes of `Person`.
- No other subtypes. Therefore, every `Person` is either `Man` or `Woman`.

Solution to an Alloy model

Solution to an Alloy model/formula is an instantiation of the domains `Man` and `Woman`, and of the relations `spouse` and `parents`:



Man = {(Man0), (Man1)}

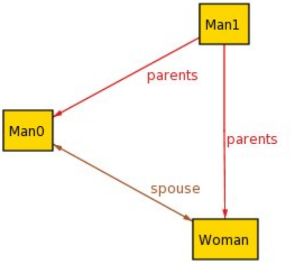
Woman = {(Woman)}

Parents = {(Man1, Man0), (Man1, Woman)}

Spouse = {(Man0, Woman), (Woman, Man0)}

Solution to an Alloy model

Solution to an Alloy model/formula is an instantiation of the domains `Man` and `Woman`, and of the relations `spouse` and `parents`:



Man = {(Man0), (Man1)}

Woman = {(Woman)}

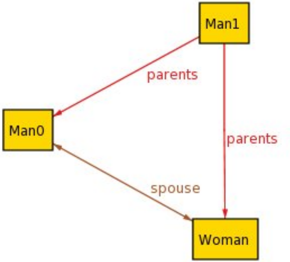
Parents = {(Man1, Man0), (Man1, Woman)}

Spouse = {(Man0, Woman), (Woman, Man0)}

"." is relational join:

$$\mathit{Man.spouse} = \{\mathit{Woman}\},\ \mathit{Woman.spouse} = \{\mathit{Man0}\}$$
$$\{\mathit{Man0}\}.\mathit{parent} = \{\},\ \{\mathit{Man1}\}.\mathit{parent} = \{\mathit{Man0},\mathit{Woman}\}$$

Key elements of Alloy model: facts



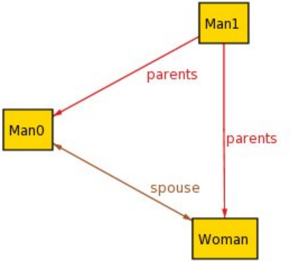
Man =
{(Man0),
(Man1)}

Woman =
{(Woman)}

Parents =
{(Man1, Man0),
(Man1, Woman)}

Spouse =
{(Man0, Woman),
(Woman, Man0)}

Key elements of Alloy model: facts



Man = {(Man0), (Man1)}

Woman = {(Woman)}

Parents = {(Man1, Man0), (Man1, Woman)}

Spouse = {(Man0, Woman), (Woman, Man0)}

The model above satisfies the following facts:

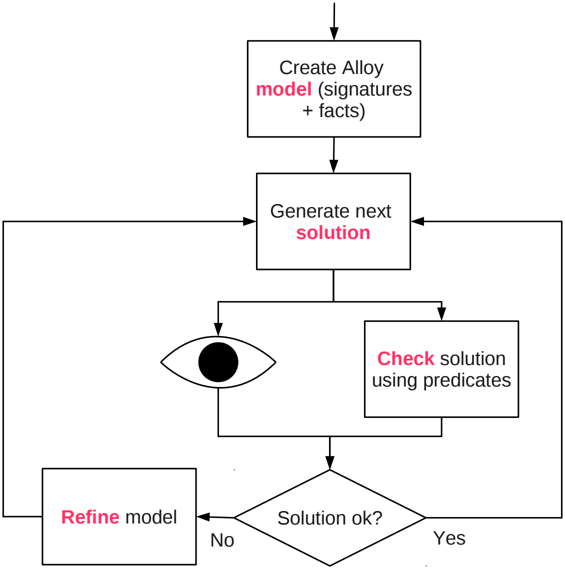
```
fact {
    spouse = ~spouse
      -- spouse is symmetric
    Man.spouse in Woman && Woman.spouse in Man
      -- a man's spouse is a woman and a woman's spouse is a man
    all p: Person | one mother: Woman | one father: Man |
        p.parents = mother + father
      -- every person has one man as a father and one woman as a mother
}
```

Alloy usage

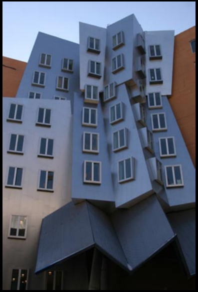
1. User supplies signatures, relations, and facts
2. Alloy generates solutions that satisfy all given facts
3. User checks if solutions are ok. If not, user corrects the facts, and goes back to Step 2.
4. User has a good model!

[*Go over genealogy1.als example*]

Typical way to use Alloy

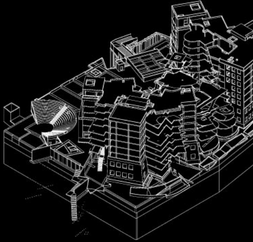


why analyzable models?



why models?
› figure out what problem you're solving
› explore invented concepts
› communicate with collaborators

why analyzable?
› not just finding errors early
› analysis breathes life into models!



xp on design models

Another strength of design with pictures is speed. In the time it would take you to code one design, you can compare and contrast three designs using pictures. The trouble with pictures, however, is that they can't give you concrete feedback… The XP strategy is that anyone can design with pictures all they want, but as soon as a question is raised that can be answered with code, the designers must turn to code for the answer. The pictures aren't saved. -- Kent Beck, Extreme Programming Explained, 2000

Signatures

```
abstract sig Person {}
sig Man, Woman extends Person {}
```

Signatures

```
abstract sig Person {}
sig Man, Woman extends Person {}
```

- Each top-level signature (e.g., `Person`) will be bound to a set of atoms that is disjoint from those bound to other top-level signatures.
- Other signatures will be bound to sets of atoms, as follows.
- Each subtype signature (e.g., `Woman`) $\subseteq$ the parent signature (e.g., `Man`)
- Immediate subtypes of any signature are disjoint from each other.
- An abstract signature is partitioned among its subtypes (`abstract` keyword is ignored if no subtypes present)
- Default cardinality of any signature is `set`. Other cardinalities allowed (e.g., "`one sig Eve ...`").
- Any signature can be used as a (set-valued) free variable in any formula.

Relations

`sig A {f:` *expr*`}`

Case *expr* is of the form "*n range*", where *range* is an expression that evaluates to a set (i.e., a unary relation), and *n* is a cardinality keyword, namely, `set`, `lone`, `some`, or `one` (default is `one`).

- `f` is basically a binary relation from `A` to *range*. That is, `f` is a subset of `A` $\times$ *range*.
- The cardinality keyword constrains the number of distinct atoms in *expr* that could be mapped to any atom in `A` by `f`.

Case *expr* is from the following grammar:

$$\begin{array}{lcl} expr & = & \textit{set-valued-expr}\ n_1 \texttt{ -> } n_2\ \textit{set-valued-expr} \\ & = & \textit{set-valued-expr}\ n_1 \texttt{ -> } n_2\ expr \end{array}$$

- Each *set-valued-expression* is an expression that evaluates to a set.
- `f` will be an $(m+1)$-ary relation, where m is the number of *set-valued-expr*s that constitute *expr*.
- Semantics of "`f: B` n_1 `->` n_2 `C`": For any element $a \in$ `A`, considering only the tuples in `f` whose first component is a, each atom from `B` is associated with n_2 atoms from `C`, and each atom from `C` is associated with n_1 atoms from `B`.

Other points about relations:

- Any relation should be defined only once (i.e., as part of the definition of some `sig`).
- Any relation can be used as a free variable in any formula. In fact, signatures, relations, and quantified variables (to be introduced later) serve as leaves in expressions/formulas.

Built-in constants

- `univ` - the universal set (full unary relation)
- `iden` - the identity relation (on the universal set)
- `none` - the empty set (empty unary relation)

Relational-valued expressions (r-exprs)

Expressions are sets

All expressions represent sets (and relations)

- Signatures, relations, quantified variables - these are leaf r-exprs
- `e1 + e2` - Union
- `e1 - e2` - Difference
- `e1 & e2` - Intersection
- `e1 -> e2` - Product
- `e1 . e2` or `e2[e1]` - Relational Join
- `e1 <: e2` - Domain restriction
- `e1 :> e2` - Range restriction
- `e1 ++ e2` - Relational override
- `~e` - Relational transpose
- `^e` - Transitive closure
- `*e` - Reflexive transitive closure

Boolean expressions

b-expr ::= r-expr [! | not] r-binOp r-expr |
  r-unOp r-expr | b-unOp b-expr | b-expr b-binOp b-expr |
  b-expr implies (*or* =>) b-expr [else b-expr]

r-binOp := in | =

r-unOp := lone | some | one | no

b-unOp ::= ! | not

b-binOp ::= || | or | && | and | <=> | iff

Formulas

formula ::= b-expr | "*n* v: r-expr | b-expr"

n ::= all | some | one | no | lone

Note

- The tool requires r-expr to be a set (i.e., unary expression)
- v is bound to individual elements of r-expr, in turn.

Models and Interpretations

- An Alloy model M is interpreted as a (conjunctive) logical formula, f_M
- Constraints enforced by signatures as well as facts automatically become part of f_M
- The predicate that's being simulated (i.e., run) becomes part of f_M.

Models and Interpretations

- An Alloy model M is interpreted as a (conjunctive) logical formula, f_M
- Constraints enforced by signatures as well as facts automatically become part of f_M
- The predicate that's being simulated (i.e., run) becomes part of f_M.
- An instance (or solution) to the model is
  - a finite universe U of atoms, and
  - an assignment of subsets of U to the different signatures, and
  - an assignment of relations (on U) to the different relations (in the `sig` declarations)

  such that it satisfies f_M.

Modeling notation $\Rightarrow$ logical formula

For example,

"`no p: Person | some p.spouse & p.parents`"

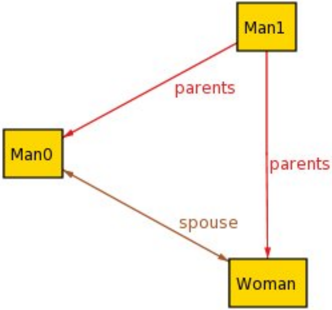
becomes

$$\nexists p \in \texttt{Person} \mid \exists q \mid \texttt{spouse}(p, q) \wedge \texttt{parents}(p, q)$$

Instance of an Alloy model

```
all p: Person | not p in p.^parents
spouse = ~spouse
Man.spouse in Woman && Woman.spouse in Man
no p: Person | some p.spouse.parents & p.parents
no p: Person | some p.spouse & p.parents
all p: Person | p.parents.spouse = p.parents
```

Solution to the Alloy model is an instantiation of the domains `Man` and `Woman`, and of the relations `spouse` and `parents`:



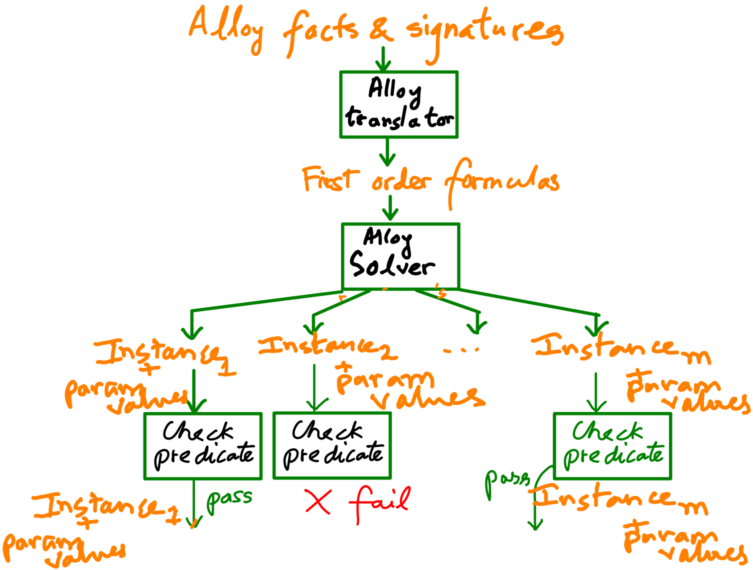
Man = {(Man0), (Man1)}

Woman = {(Woman)}

Parents = {(Man1, Man0), (Man1, Woman)}

Spouse = {(Man0, Woman), (Woman, Man0)}

Alloy architecture



Commands

- “ `run P`”: Finds instances of the Alloy model that satisfy all facts and that satisfy predicate `P`.
  If `P` has a formal parameter f, it finds pairs (m, f_V) that satisfy all facts as well as predicate `P`, where m is an instance and f_V is a valuation for f (i.e., a set/relation based on the atoms in m).
- “`check P`”: semantically equivalent to “`run (not P)`”.

Why Alloy does not use first-order logic

- Obvious strategy would have been to directly apply theorem-prover on first-order formulas (i.e., formulas with quantifiers)
- There are difficulties with this approach
  - Most versions of the problem are undecidable.
    - Therefore, theorem provers typically need human assistance (e.g., user-provided invariants)
  - Theorem provers are not efficient.
  - They may not provide instances for satisfiable formulas (i.e., they may just declare the formula to be satisfiable or unsatifiable)
    - Generating instances is important for human users, because conceptual models are usually buggy to begin with. Users find it easy to refine models by studying instances.

Alloy's approach: translate model to a propositional formula

- Satisfiability checking on proposition formulas (i.e., SAT) is decidable.
- Although problem is NP-Complete, modern implementations are scalable.
- They provide solutions (i.e., instances) for satisfiable formulas.
- They can provide unsatisfiable cores (i.e., a small subset of the formula that is unsatisfiable) for unsatisfiable formulas. This aids human understanding.

High-level idea behind translation

- Size n_S of each top-level signature S is an input parameter to the translation.
- For each lower-level signature g that is derived from a top-level signature S, we introduce boolean variables $g_0, g_1, \ldots, g_{n_S - 1}$.
  - Meaning: g_i is true in solution means ith element of S belongs to g.
- For each declared relation r on the signature $S_1 \times S_2$, we introduce boolean variables $\{r_{ij} \mid 0 \leq i < n_{S_1}, 0 \leq j < n_{S_2}\}$.
  - Meaning: r_{ij} is true in solution means $(S_1[i], S_2[j]) \in r$.
- Theorem: Assume there is a single top-level signature S. A first-order formula f has an instance with n atoms *iff* the translated formula obtained from f using n as the translation-parameter has a corresponding instance.