From:

*FM 2006 Alloy Intro and Logic*

Greg Dennis and Rob Seater
Software Design Group, MIT
agenda

- Intro & Logic
- Language & Analysis
- Dynamic Modeling
Observations

- Software is built on abstractions
  - Pick the right ones, and programming will flow naturally from design.
    - an idea reduced to its essential form
  - Pick the wrong ones, and programming will be a series of nasty surprises.

- Our current strategy does not guarantee that the designs are correct.

- Formal specifications help to verify correctness but are hard to read and write.

\[
\text{send:}\quad \text{The program } \text{send}(e_1, e_2).P \text{ sends a message with contents } e_2 \text{ to the actor indicated by } e_1:\
\]

\[
a[\text{send}(e_1, e_2).P]_s \xrightarrow{\tau} a[P]_s | \langle [e_1]_s^a, [e_2]_s^a \rangle
\]

where \(\tau\) represents an internal invisible step of computation.
New Approach - Alloy

- Alloy takes from *formal specification* the idea of precise and expressive notation based on a tiny core of simple and robust concepts.
- It replaces conventional analysis based on *theorem proving* with a fully automatic analysis.
- Analysis is not complete but examines a finite space of cases (configurable)
  - NO TEST CASES ARE REQUIRED
  - User specifies predicates to be checked
four key ideas . . .

1) everything is a relation

2) non-specialized logic

3) counterexamples & scope

4) analysis by SAT
1) everything's a relation

- Alloy uses relations for
  - all datatypes – even sets, scalars, tuples
  - structures in space and time

- key operator is **dot** join
  - relational join
  - field navigation
  - ...
why relations?

- easy to understand
  - binary relation is a graph or mapping
- easy to analyze
  - first order (tractable)
- uniform

set of addresses associated with name n in set of books B
Alloy: n.(B.addr)
Z: \( \cup \{ b: B \cdot b.addr (| \{n\} |) \}\)
OCL: B.addr[n]->asSet()

There is no problem in computer science that cannot be solved by an extra level of indirection.
  – David Wheeler
2) non-specialized logic

- No special constructs for state machines, traces, synchronization, concurrency . . .
3) counterexamples & scope

- observations about design analysis:
  - most assertions are wrong
  - most flaws have small counterexamples

![Diagram showing testing and scope-complete cases]

- testing: a few cases of arbitrary size
- scope-complete: all cases within a small bound
4) analysis by SAT

- SAT, the quintessential hard problem (Cook 1971)
  - SAT is hard, so reduce SAT to your problem
- SAT, the universal constraint solver (Kautz, Selman, ... 1990's)
  - SAT is easy, so reduce your problem to SAT
  - solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), ...

Stephen Cook  Eugene Goldberg  Henry Kautz  Sharad Malik  Yakov Novikov
The **Boolean satisfiability problem (SAT)** is a decision problem considered in complexity theory. An instance of the problem is a Boolean expression written using only **AND**, **OR**, **NOT**, variables, and parentheses. The question is: given the expression, is there some assignment of **TRUE** and **FALSE** values to the variables that will make the entire expression true?

\[(A \lor B \lor \neg B) \land (D \lor B) \ldots\]
Alloy analyzer architecture
Try the Alloy Analyzer

- requires Java
  - http://java.sun.com

- download the Alloy Analyzer
  - http://alloy.mit.edu

- run the Analyzer
  - double click alloy.jar or
  - execute java -jar alloy.jar at the command line
Example: modeling “ceilings and floors”

sig Platform {}
there are “Platform” things

sig Man {ceiling, floor: Platform}
each Man has a ceiling and a floor Platform

pred Above(m, n: Man) {m.floor = n.ceiling}
Man m is “above” Man n if m's floor is n's ceiling

fact {all m: Man | some n: Man | Above (n,m)}
"One Man's Ceiling Is Another Man's Floor"
checking “ceilings and floors”

```plaintext
assert BelowToo {  
  all m: Man | some n: Man | Above (m,n)  
}

"One Man's Floor Is Another Man's Ceiling"?

check BelowToo for 2

check "One Man's Floor Is Another Man's Ceiling"

counterexample with 2 or less platforms and men?

• clicking “Execute” ran this command
  – counterexample found, shown in graphic
counterexample to “BelowToo”

McNaughton
Alloy = logic + language + analysis

• logic
  – first order logic + relational calculus

• language
  – syntax for structuring specifications in the logic

• analysis
  – bounded exhaustive search for counterexample to a claimed property using SAT
logic: relations of atoms

- atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted

- relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms

- every value in Alloy logic is a relation!
  - relations, sets, scalars all the same thing
logic: everything's a relation

- sets are unary (1 column) relations
  
  \[
  \begin{align*}
  \text{Name} &= \{(N0), (N1), (N2)\}, \\
  \text{Addr} &= \{(A0), (A1), (A2)\}, \\
  \text{Book} &= \{(B0), (B1)\}
  \end{align*}
  \]

- scalars are singleton sets

  \[
  \begin{align*}
  \text{myName} &= \{(N1)\}, \\
  \text{yourName} &= \{(N2)\}, \\
  \text{myBook} &= \{(B0)\}
  \end{align*}
  \]

- binary relation

  \[
  \text{names} = \{(B0, N0), (B0, N1), (B1, N2)\}
  \]

- ternary relation

  \[
  \text{addrs} = \{(B0, N0, A0), (B0, N1, A1), (B1, N1, A2), (B1, N2, A2)\}
  \]
logic: relations

\[
\text{addr} = \{(B0, N0, A0), (B0, N1, A1), (B1, N1, A2), (B1, N2, A2)\}
\]

<table>
<thead>
<tr>
<th>B0</th>
<th>N0</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>N1</td>
<td>A1</td>
</tr>
<tr>
<td>B1</td>
<td>N1</td>
<td>A2</td>
</tr>
<tr>
<td>B1</td>
<td>N2</td>
<td>A2</td>
</tr>
</tbody>
</table>

\[\text{size} = 4\]
\[\text{arity} = 3\]

- rows are unordered
- columns are ordered but unnamed
- all relations are first-order
  - relations cannot contain relations, no sets of sets
logic: address book example

Name = \{ (N0), (N1), (N2) \}
Addr = \{ (A0), (A1), (A2) \}
Target = \{ (N0), (N1), (N2), (A0), (A1), (A2) \}
address = \{ (N0, A1), (N1, N2), (N2, A1), (N2, A0) \}
logic: constants

<table>
<thead>
<tr>
<th>none</th>
<th>empty set</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ</td>
<td>universal set</td>
</tr>
<tr>
<td>iden</td>
<td>identity relation</td>
</tr>
</tbody>
</table>

Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1)}

none = {}
univ = {(N0), (N1), (N2), (A0), (A1)}
iden = {(N0, N0), (N1, N1), (N2, N2), (A0, A0), (A1, A1)}
logic: set operators

+ union
& intersection
- difference
in subset
= equality

Name = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed = {(N2)}
Name - RecentlyUsed = {(N1)}
RecentlyUsed in Alias = false
RecentlyUsed in Name = true
Name = Group + Alias = true

greg = {(N0)}
rob = {(N1)}
greg + rob = {(N0), (N1)}
greg = rob = false
rob in none = false

cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}
cacheAddr + diskAddr =
cacheAddr & diskAddr =
cacheAddr = diskAddr =
logic: product operator

-> cross product

Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

Name->Addr = {(N0, A0), (N0, A1), (N1, A0), (N1, A1)}

b = {(B0)}
b' = {(B1)}

b->b' = 

Book->Name->Addr = 
{(B0, N0, A0), (B0, N1, A0), (B1, N0, A0), (B1, N1, A0)}

address = {(N0, A0), (N1, A1)}
address' = {(N2, A2)}

b->address + b'->address' =

b->address + b'->address' = {(B0, N0, A0), (B0, N1, A0), (B1, N2, A2)}
logic: relational join

\[ p \cdot q \equiv \]
\[ \begin{array}{c}
\langle a, b \rangle \\
\langle a, c \rangle \\
\langle b, d \rangle
\end{array} \]
\[ = \]
\[ \begin{array}{c}
\langle a, d, c \rangle \\
\langle b, c, c \rangle \\
\langle c, c, c \rangle \\
\langle b, a, d \rangle
\end{array} \]
\[ = \]
\[ \langle a, c, c \rangle \\
\langle a, a, d \rangle \]

\[ x.f \equiv \]
\[ x \]
\[ \begin{array}{c}
\langle c \rangle \\
\langle b, d \rangle \\
\langle c, a \rangle \\
\langle d, a \rangle
\end{array} \]
\[ = \]
\[ \begin{array}{c}
\langle a \rangle
\end{array} \]
logic: join operators

. dot join
[] box join

\[ e_1[e_2] = e_2.e_1 \]
\[ a.b.c[d] = d.(a.b.c) \]

Book = \{(B0)\}
Name = \{(N0), (N1), (N2)\}
Addr = \{(A0), (A1), (A2)\}
Host = \{(H0), (H1)\}

myName = \{(N1)\}
myAddr = \{(A0)\}

address = \{(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)\}
host = \{(A0, H0), (A1, H1), (A2, H1)\}

Book.address = \{(N0, A0), (N1, A0), (N2, A2)\}
Book.address[myName] = \{(A0)\}
Book.address.myName = \{\}

host[myAddr] = \{(H0)\}
address.host = \{(B0, N0, H0), (B0, N1, H0), (B0, N2, H1)\}
logic: unary operators

\[\sim\] transpose
\[\wedge\] transitive closure
\[\ast\] reflexive transitive closure

apply only to binary relations

\[\wedge r = r + r.r + r.r.r + \ldots\]
\[\ast r = \text{idem} + \wedge r\]

Node = \{(N0), (N1), (N2), (N3)\}
next = \{(N0, N1), (N1, N2), (N2, N3)\}

\sim next = \{(N1, N0), (N2, N1), (N3, N2)\}
\wedge next = \{(N0, N1), (N0, N2), (N0, N3),
(N1, N2), (N1, N3),
(N2, N3)\}
\ast next = \{(N0, N0), (N0, N1), (N0, N2), (N0, N3),
(N1, N1), (N1, N2), (N1, N3),
(N2, N2), (N2, N3), (N3, N3)\}

first = \{(N0)\}
rest = \{(N1), (N2), (N3)\}

first.\wedge next = rest
first.\ast next = \text{Node}
logic: restriction and override

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;:</td>
<td>domain restriction</td>
</tr>
<tr>
<td>&gt;:</td>
<td>range restriction</td>
</tr>
<tr>
<td>++</td>
<td>override</td>
</tr>
</tbody>
</table>

\[
p \text{ ++ } q = p - (\text{domain}(q) <: p) + q\]

Name = \{(N0), (N1), (N2)\}
Alias = \{(N0), (N1)\}
Addr = \{(A0)\}
address = \{(N0, N1), (N1, N2), (N2, A0)\}

\[
\text{address} :> \text{Addr} = \{(N2, A0)\}
\]

Alias <: address = address :> Name = \{(N0, N1), (N1, N2)\}
address :> Alias = \{(N0, N1)\}

workAddress = \{(N0, N1), (N1, A0)\}
address ++ workAddress = \{(N0, N1), (N1, A0), (N2, A0)\}

The preferred for an alias, which is the workAddress if it exists, and otherwise the homeAddress.
logic: boolean operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!</code></td>
<td>not</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>and</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>=&gt;</code></td>
<td>implies</td>
</tr>
<tr>
<td><code>,</code></td>
<td>else</td>
</tr>
<tr>
<td><code>&lt;=&gt;</code></td>
<td>iff</td>
</tr>
</tbody>
</table>

four equivalent constraints:

F => G , H

F implies G else H

(F && G) || (!F) && H

(F and G) or (not F) and H
# logic: quantifiers

<table>
<thead>
<tr>
<th>All</th>
<th>Some</th>
<th>No</th>
<th>Lone</th>
<th>One</th>
</tr>
</thead>
<tbody>
<tr>
<td>all $x: e</td>
<td>F</td>
<td>all</td>
<td>F holds for every $x$ in $e$</td>
<td></td>
</tr>
<tr>
<td>all $x: e_1, y: e_2</td>
<td>F</td>
<td>some</td>
<td>F holds for at least one $x$ in $e$</td>
<td></td>
</tr>
<tr>
<td>all $x, y: e</td>
<td>F</td>
<td>no</td>
<td>F holds for no $x$ in $e$</td>
<td></td>
</tr>
<tr>
<td>all disj $x, y: e</td>
<td>F</td>
<td>lone</td>
<td>F holds for at most one $x$ in $e$</td>
<td></td>
</tr>
<tr>
<td>all $n: Name, a: Address</td>
<td>a in n.address</td>
<td>one</td>
<td>F holds for exactly one $x$ in $e$</td>
<td></td>
</tr>
</tbody>
</table>

- **some** $n: Name, a: Address | a in n.address
  
some name maps to some address — address book not empty

- **no** $n: Name | n in n.^address

- **all** $n: Name | lone $a: Address | a in n.address

- **all** $n: Name | no disj $a, a': Address | (a + a') in n.address

- **all** $x: e | F
- **all** $x, y: e | F
- **all disj** $x, y: e | F

- **some** $n: Name, a: Address | a in n.address

- **no** $n: Name | n in n.^address

- **all** $n: Name | lone $a: Address | a in n.address

- **all** $n: Name | no disj $a, a': Address | (a + a') in n.address
logic: quantified expressions

| some e   | e has at least one tuple |
| no e     | e has no tuples          |
| lone e   | e has at most one tuple  |
| one e    | e has exactly one tuple  |

some Name
set of names is not empty

some address
address book is not empty – it has a tuple

no (address.Addr - Name)
nothing is mapped to addresses except names

all n: Name | lone n.address
every name maps to at most one address
logic: comprehensions

\{x_1: e_1, x_2: e_2, \ldots, x_n: e_n \mid F\}

\{n: \text{Name} \mid \textbf{no} \ n. \wedge \text{address} \& \text{Addr}\}

set of names that don't resolve to any actual addresses

\{n: \text{Name}, a: \text{Address} \mid n \rightarrow a \ \textbf{in} \ \wedge \text{address}\}

binary relation mapping names to reachable addresses
logic: if and let

\[
\text{if } f \text{ then } e_1 \text{ else } e_2
\]
\[
\text{let } x = e \mid \text{constraint}
\]
\[
\text{let } x = e \mid \text{expression}
\]

four equivalent constraints:

\[
\text{all } n: \text{Name} \mid
\begin{align*}
\text{some } n.\text{workAddress} & \Rightarrow n.\text{address} = n.\text{workAddress} \\
\text{else } n.\text{address} & = n.\text{homeAddress}
\end{align*}
\]

\[
\text{all } n: \text{Name} \mid
\begin{align*}
\text{let } w = n.\text{workAddress}, a = n.\text{address} \mid
\text{some } w & \Rightarrow a = w \text{ else } a = n.\text{homeAddress}
\end{align*}
\]

\[
\text{all } n: \text{Name} \mid
\begin{align*}
\text{let } w = n.\text{workAddress} \mid
n.\text{address} & = \text{if some } w \text{ then } w \text{ else } n.\text{homeAddress}
\end{align*}
\]

\[
\text{all } n: \text{Name} \mid
\begin{align*}
n.\text{address} & = \text{let } w = n.\text{workAddress} \mid
\text{if some } w \text{ then } w \text{ else } n.\text{homeAddress}
\end{align*}
\]
logic: cardinalities

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#r</td>
<td>number of tuples in r</td>
</tr>
<tr>
<td>0, 1, ...</td>
<td>integer literal</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equals</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

**sum** x: e | ie

sum of integer expression ie for all values of scalar x drawn from e

**all** b: Bag | #b.marbles <= 3

all bags have 3 or less marbles

#Marble = **sum** b: Bag | #b.marbles

the sum of the marbles across all bags equals the total number of marbles
2 logics in one

• “everybody loves a winner”

• predicate logic
  - $\forall w \mid \text{Winner}(w) \implies \forall p \mid \text{Loves}(p, w)$

• relational calculus
  - $\text{Person} \times \text{Winner} \subseteq \text{loves}$

• Alloy logic – any way you want
  - all $p: \text{Person}, w: \text{Winner} \mid p \rightarrow w$ in loves
  - Person $\rightarrow$ Winner in loves
  - all $p: \text{Person} \mid \text{Winner}$ in $p$.loves