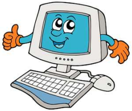
Advancing Declarative Programming

Aleksandar Milicevic
Massachusetts Institute of Technology

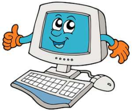
May 07, 2015

What is **Declarative** Programming?

What is **Declarative** Programming?

- say **what** , not how 
  - describe what the program is intended to do in some terms that are both expressive and easy to use

What is **Declarative** Programming?

- say **what**  , not how 
  - describe what the program is intended to do in some terms that are both expressive and easy to use
- *"It would be very nice to input this description into some suitably programmed computer, and get the computer to translate it automatically into a subroutine"*

  \- C. A. R. Hoare ["An overview of some formal methods for program design", 1987]

Spectrum of The Declarative Programming Space

spec	formal logic		**spec**	DSL
engine	sophisticated search		**engine**	translation/compilation
apps	complex algorithms, constraint solving		**apps**	domain-specific uses

Spectrum of The Declarative Programming Space

(my previous work)

executable specs for java

program synthesis

Squander

Jennisys

spec	formal logic
engine	sophisticated search
apps	complex algorithms, constraint solving

spec	DSL
engine	translation/compilation
apps	domain-specific uses

Spectrum of The Declarative Programming Space

(my previous work)

executable specs for java

program synthesis

spec	formal logic
engine	sophisticated search
apps	complex algorithms, constraint solving

spec	DSL
engine	translation/compilation
apps	domain-specific uses

ALLOY* [ABZ'12,SCP'14,ICSE'15]

- more powerful constraint solver
- capable of solving a whole new category of formal specifications

Spectrum of The Declarative Programming Space

(my previous work)

executable specs for java — Squander

program synthesis — Jennisys

spec formal logic
engine sophisticated search
apps complex algorithms, constraint solving

ALLOY* [ABZ'12,SCP'14,ICSE'15]

- more powerful constraint solver
- capable of solving a whole new category of formal specifications

spec DSL
engine translation/compilation
apps domain-specific uses

SUNNY [Onward'13]

- model-based web framework
- reactive, single-tier, policy-agnostic
- what instead of how

Spectrum of The Declarative Programming Space

(my previous work)

executable specs for java — Squander

program synthesis — Jennisys

spec formal logic
engine sophisticated search
apps complex algorithms, constraint solving

spec DSL
engine translation/compilation
apps domain-specific uses

ARbY [ABZ'14]

• unified specification & implementation language

ALLOY* [ABZ'12,SCP'14,ICSE'15]

• more powerful constraint solver
• capable of solving a whole new category of formal specifications

SUNNY [Onward'13]

• model-based web framework
• reactive, single-tier, policy-agnostic
• what instead of how

ALLOY*: Higher-Order Constraint Solving

(my previous work)

executable specs for java

Squander

program synthesis

Jennisys

spec formal logic
engine sophisticated search
apps complex algorithms, constraint solving

spec DSL
engine translation/compilation
apps domain-specific uses

ARbY [ABZ’14]

- unified specification & implementation language

ALLOY* ABZ’12,SCP’14,ICSE’15]

- more powerful constraint solver
- capable of solving a whole new category of formal specifications

SUNNY [Onward’13]

- model-based web framework
- reactive, single-tier, policy-agnostic
- what instead of how

What is ALLOY*

ALLOY*: **a more powerful version of the alloy analyzer**

What is ALLOY*

ALLOY*: a more powerful version of the alloy analyzer

typical uses of the alloy analyzer

• bounded software verification	→	but no software synthesis
• analyze safety properties of event traces	→	but no liveness properties
• find a safe full configuration	→	but not a safe partial conf
• find an instance satisfying a property	→	but no min/max instance

What is ALLOY*

ALLOY*: a more powerful version of the alloy analyzer

typical uses of the alloy analyzer

• bounded software verification	→	but no software synthesis
• analyze safety properties of event traces	→	but no liveness properties
• find a safe full configuration	→	but not a safe partial conf
• find an instance satisfying a property	→	but no min/max instance

$\underbrace{\text{but no software synthesis, \ldots, but no min/max instance}}_{\text{higher-order}}$

What is ALLOY*

ALLOY*: a more powerful version of the alloy analyzer

typical uses of the alloy analyzer

- bounded software verification → but no software synthesis
- analyze safety properties of event traces → but no liveness properties
- find a safe full configuration → but not a safe partial conf
- find an instance satisfying a property → but no min/max instance

higher-order

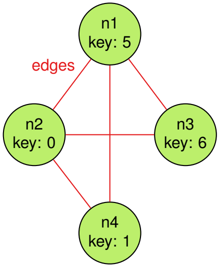
ALLOY*

- capable of automatically solving arbitrary higher-order formulas

First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

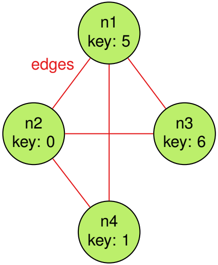
- every two nodes in a clique must be connected



First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

- every two nodes in a clique must be connected

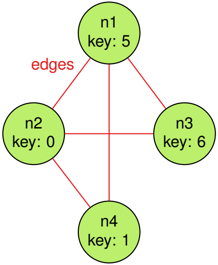



```
sig Node { key: one Int }
```

First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

- every two nodes in a clique must be connected

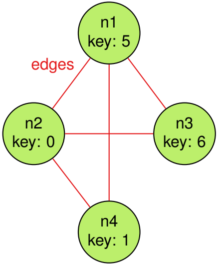



```
sig Node { key: one Int }

run {
  some edges: Node -> Node |
    some clq: set Node |
      clique[edges, clq]
}
```

First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

- every two nodes in a clique must be connected

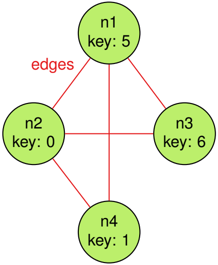



```
sig Node { key: one Int }

run {
  some edges: Node -> Node |
    some clq: set Node |
      clique[edges, clq]
}

pred clique[edges: Node->Node, clq: set Node] {
  all disj n1, n2: clq | n1->n2 in edges
}
```

First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

- every two nodes in a clique must be connected



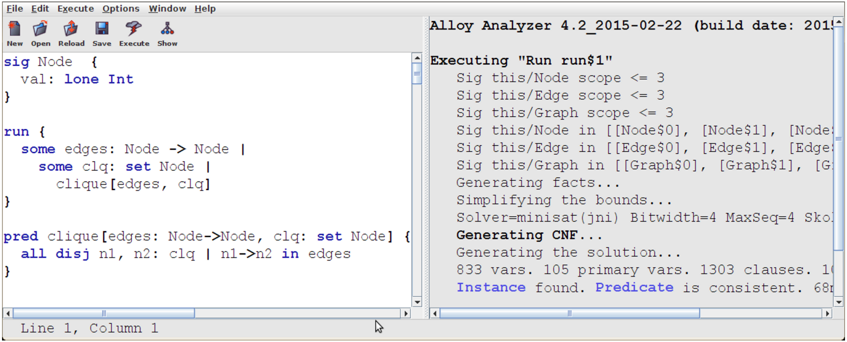

```
sig Node { key: one Int }

run {
  some edges: Node -> Node |
    some clq: set Node |
      clique[edges, clq]
}

pred clique[edges: Node->Node, clq: set Node] {
  all disj n1, n2: clq | n1->n2 in edges
}
```

- **Alloy Analyzer**: automatic, bounded, relational constraint solver

First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

- every two nodes in a clique must be connected

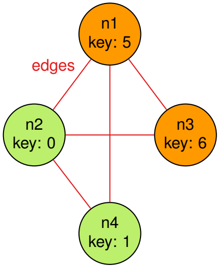


- **Alloy Analyzer**: automatic, bounded, relational constraint solver
- a **solution** (automatically found by Alloy): **clq** = $\{n_1, n_3\}$

First-Order Vs. Higher-Order: `clique`

first-order: finding a graph and a clique in it

- every two nodes in a clique must be connected




```
sig Node { key: one Int }

run {
  some edges: Node -> Node |
    some clq: set Node |
      clique[edges, clq]
}

pred clique[edges: Node->Node, clq: set Node] {
  all disj n1, n2: clq | n1->n2 in edges
}
```

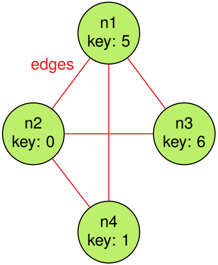
- **Alloy Analyzer**: automatic, bounded, relational constraint solver

- a **solution** (automatically found by Alloy): **clq** = $\{n_1, n_3\}$

First-Order Vs. **Higher**-Order: `maxClique`

higher-order: finding a graph and a maximal clique in it

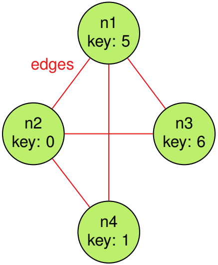
- there is no other clique with more nodes



First-Order Vs. **Higher**-Order: `maxClique`

higher-order: finding a graph and a maximal clique in it

- there is no other clique with more nodes




```
pred maxClique[edges: Node->Node,  clq: set Node] {
  clique[edges, clq]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}

run {
  some edges: Node -> Node |
    some clq: set Node |
      maxClique[edges, clq]
}
```

First-Order Vs. **Higher**-Order: `maxClique`

higher-order: finding a graph and a maximal clique in it

- there is no other clique with more nodes

expressible but <u>not solvable</u> in Alloy!

File Edit Execute Options Window Help

New Open Reload Save Execute Show

```
sig Node { key: Int }

pred clique[edges: Node->Node, clq: set Node] {
  all disj n1, n2: clq | n1->n2 in edges
}

pred maxClique[edges: Node->Node, clq: set Node] {
  clique[edges, clq]
  all ns: set Node |
    not (clique[edges, ns] and #ns > #clq)
}

run { // find a maximal clique in a given graph
  let edges = Node -> Node |
  some clq: set Node | maxClique[edges, clq]
}
```

Line 10, Column 7

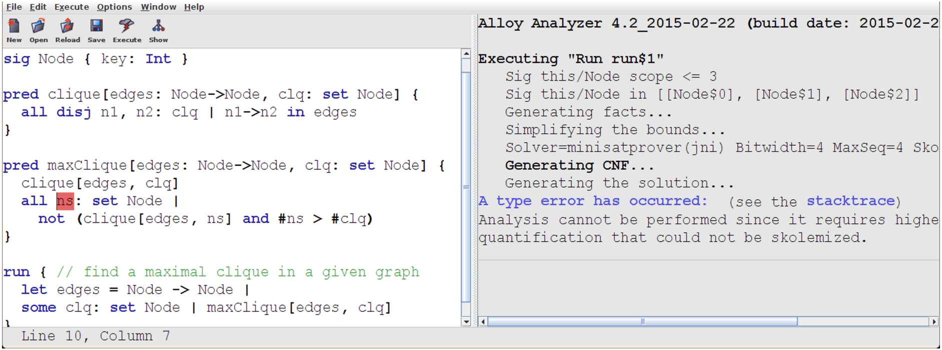
```
Alloy Analyzer 4.2_2015-02-22 (build date: 2015-02-2

Executing "Run run$1"
   Sig this/Node scope <= 3
   Sig this/Node in [[Node$0], [Node$1], [Node$2]]
   Generating facts...
   Simplifying the bounds...
   Solver=minisatprover(jni) Bitwidth=4 MaxSeq=4 Sko
   Generating CNF...
   Generating the solution...
A type error has occurred:  (see the stacktrace)
Analysis cannot be performed since it requires highe
quantification that could not be skolemized.
```

First-Order Vs. **Higher**-Order: `maxClique`

higher-order: finding a graph and a maximal clique in it

- there is no other clique with more nodes

expressible but <u>not solvable</u> in Alloy!



- **definition** of higher-order (as in Alloy):
  - quantification over all sets of atoms

Solving `maxClique` Vs. Program **Synthesis**

program synthesis	**maxClique**
find some program AST s.t., for all possible values of its inputs its specification holds	find some set of nodes s.t., it is a clique and for all possible other sets of nodes not one is a larger clique

program synthesis

```
some program: ASTNode |
  all env: Var -> Val |
    spec[program, env]
```

maxClique

```
some clq: set Node |
  clique[clq] and
  all ns: set Node |
    not (clique[ns] and #ns > #clq)
```

Solving `maxClique` Vs. Program **Synthesis**

program synthesis	**maxClique**
find <u>some</u> program AST s.t., for <u>all</u> possible values of its inputs its specification holds	find <u>some</u> set of nodes s.t., it is a clique and for <u>all</u> possible other sets of nodes not one is a larger clique
`some program: ASTNode | all env: Var -> Val | spec[program, env]`	`some clq: set Node | clique[clq] and all ns: set Node | not (clique[ns] and #ns > #clq)`

similarities:

- the same **`some`**/**`all`** (∃∀) pattern
- the **`all`** quantifier is higher-order

Solving `maxClique` Vs. Program **Synthesis**

program synthesis	**maxClique**
find <u>some</u> program AST s.t., for <u>all</u> possible values of its inputs its specification holds	find <u>some</u> set of nodes s.t., it is a clique and for <u>all</u> possible other sets of nodes not one is a larger clique
`some program: ASTNode | all env: Var -> Val | spec[program, env]`	`some clq: set Node | clique[clq] and all ns: set Node | not (clique[ns] and #ns > #clq)`

```
some program: ASTNode |
  all env: Var -> Val |
    spec[program, env]
```

```
some clq: set Node |
  clique[clq] and
  all ns: set Node |
    not (clique[ns] and #ns > #clq)
```

similarities:

- the same **`some`**/**`all`** (∃∀) pattern
- the **`all`** quantifier is higher-order

how do existing program synthesizers work?

CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

```
run { some prog: ASTNode | all env: Var -> Val | spec[prog, env] }
```

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

```
run { some prog: ASTNode | all env: Var -> Val | spec[prog, env] }
```

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

1. search: find *some* program and *some* environment s.t. the spec holds, i.e.,

   ```
   run { some prog: ASTNode | some env: Var -> Val | spec[prog, env] }
   ```

   to get a concrete *candidate* program `$prog`

CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

```
run { some prog: ASTNode | all env: Var -> Val | spec[prog, env] }
```

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

1. search: find *some* program and *some* environment s.t. the spec holds, i.e.,
   ```
   run { some prog: ASTNode | some env: Var -> Val | spec[prog, env] }
   ```
   to get a concrete *candidate* program `$prog`

2. verification: check if `$prog` holds for *all* possible environments:
   ```
   check { all env: Var -> Val | spec[$prog, env] }
   ```
   Done if verified; else, a concrete *counterexample* `$env` is returned as witness.

CEGIS: A Common Approach for Program Synthesis

original synthesis formulation

```
run { some prog: ASTNode | all env: Var -> Val | spec[prog, env] }
```

Counter-Example Guided Inductive Synthesis [Solar-Lezama, ASPLOS'06]

1. search: find *some* program and *some* environment s.t. the spec holds, i.e.,
   ```
   run { some prog: ASTNode | some env: Var -> Val | spec[prog, env] }
   ```
   to get a concrete *candidate* program `$prog`

2. verification: check if `$prog` holds for *all* possible environments:
   ```
   check { all env: Var -> Val | spec[$prog, env] }
   ```
   Done if verified; else, a concrete *counterexample* `$env` is returned as witness.

3. induction: *incrementally* find a new program that *additionally* satisfies `$env`:
   ```
   run { some prog: ASTNode |
             some env: Var -> Val | spec[prog, env] and spec[prog, $env]}
   ```
   If UNSAT, return no solution; else, go to 2.

ALLOY* key insight

CEGIS can be applied to solve **arbitrary higher-order** formulas

ALLOY*

generality

- solve arbitrary higher-order formulas
- no domain-specific knowledge needed

ALLOY*

generality

- solve arbitrary higher-order formulas
- no domain-specific knowledge needed

implementability

- key solver features for efficient implementation:
  - *partial instances*
  - *incremental solving*

ALLOY*

generality

- solve arbitrary higher-order formulas
- no domain-specific knowledge needed

implementability

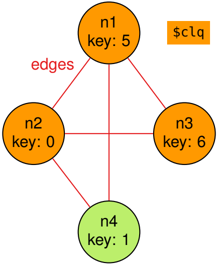
- key solver features for efficient implementation:
  - *partial instances*
  - *incremental solving*

wide applicability (in contrast to specialized synthesizers)

- program synthesis: `SyGuS` benchmarks
- security policy synthesis: `Margrave`
- solving graph problems: `max-cut`, `max-clique`, `min-vertex-cover`
- bounded verification: `Turán's` theorem

Generality: Nested Higher-Order Quantifiers

```
fun keysum[nodes: set Node]: Int {
  sum n: nodes | n.key
}

pred maxMaxClique[edges: Node->Node, clq: set Node] {
  maxClique[edges, clq]
  all ns: set Node |
    not (maxClique[edges,clq2] and
         keysum[ns] > keysum[clq])
}

run maxMaxClique for 5
```




```
Executing "Run maxMaxClique for 5"
   Solver=minisat(jni) Bitwidth=5 MaxSeq=5 SkolemDepth=3 Symmetry=20
   13302 vars. 831 primary vars. 47221 clauses. 66ms.
   Solving...
   [Some4All] started (formula, bounds)
   [Some4All]  candidate found (candidate)
   [Some4All]   verifying candidate (condition, pi) counterexample
                |- [OR] solving splits (formula)
                |- [OR]   trying choice (formula, bounds) unsat
                |- [OR]   trying choice (formula, bounds) instance
                |-   [Some4All] started (formula, bounds)
                |-   [Some4All]  candidate found (candidate)
                |-   [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
   [Some4All]   searching for next candidate  (increment)
   [Some4All]  candidate found (candidate)
   [Some4All]   verifying candidate (condition, pi) counterexample
                |- [OR] solving splits (formula)
                |- [OR]   trying choice (formula, bounds) unsat
                |- [OR]   trying choice (formula, bounds) instance
                |-   [Some4All] started (formula, bounds)
                |-   [Some4All]  candidate found (candidate)
                |-   [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
   [Some4All]   searching for next candidate  (increment)
   [Some4All]  candidate found (candidate)
   [Some4All]   verifying candidate (condition, pi) success (#cand = 3)
                |- [OR] solving splits (formula)
                |- [OR]   trying choice (formula, bounds) unsat
                |- [OR]   trying choice (formula, bounds) unsat
                |-   [Some4All] started (formula, bounds)
   Instance found. Predicate is consistent. 490ms.
```

Semantics: General Idea

- CEGIS: defined only for a single idiom (the $\exists\forall$ formula pattern)

Semantics: General **Idea**

- CEGIS: defined only for a single idiom (the $\exists\forall$ formula pattern)
- ALLOY*: generalized to arbitrary formulas

Semantics: General **Idea**

- CEGIS: defined only for a single idiom (the $\exists\forall$ formula pattern)
- ALLOY*: generalized to arbitrary formulas
  1. perform standard transformation: NNF and skolemization

Semantics: General **Idea**

- CEGIS: defined only for a single idiom (the $\exists\forall$ formula pattern)
- ALLOY*: generalized to arbitrary formulas
  1. perform standard transformation: NNF and skolemization
  2. decompose arbitrary formula into known idioms
     - → `FOL` : first-order formula
     - → `OR` : disjunction
     - → `∃∀` : higher-order top-level $\forall$ quantifier (not skolemizable)

Semantics: General **Idea**

- CEGIS: defined only for a single idiom (the $\exists\forall$ formula pattern)
- ALLOY*: generalized to arbitrary formulas
  1. perform standard transformation: NNF and skolemization
  2. decompose arbitrary formula into known idioms
     - → `FOL` : first-order formula
     - → `OR` : disjunction
     - → `∃∀` : higher-order top-level $\forall$ quantifier (not skolemizable)
  3. solve using the following decision procedure
     - → `FOL` : solve directly with Kodkod (first-order relational solver)
     - → `OR` : solve each disjunct separately
     - → `∃∀` : apply CEGIS

ALLOY* Implementation **Caveats**

```
some prog: Node |
  acyclic[prog]
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

$\rightarrow$

$\exists\forall($*conj*: `$prog in Node and acyclic[$prog],`
eQuant: `some eval ...,`
aQuant: `all eval ...)`

ALLOY* Implementation **Caveats**

```
some prog: Node |
  acyclic[prog]
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

$\rightarrow$

$\exists\forall($*conj*: `$prog in Node and acyclic[$prog],`
eQuant: `some eval ...,`
aQuant: `all eval ...`$)$

1. candidate search

- `solve` *conj* $\wedge$ *eQuant*

$\rightarrow$ *candidate instance* `$cand`: values of all relations except *eQuant.var*

ALLOY* Implementation **Caveats**

```
some prog: Node |
  acyclic[prog]
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

$\rightarrow$

$\exists\forall($*conj*: `$prog in Node and acyclic[$prog]`,
eQuant: `some eval ...`,
aQuant: `all eval ...`)

1. candidate search

- `solve` $conj \wedge eQuant$

$\rightarrow$ *candidate instance* `$cand`: values of all relations except $eQuant.var$

2. verification

- `solve` $\neg aQuant$ against the `$cand` *partial instance*

$\rightarrow$ *counterexample* `$cex`: value of the $eQuant.var$ relation

ALLOY* Implementation **Caveats**

```
some prog: Node |
  acyclic[prog]
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

$\rightarrow$ $\exists\forall(conj$: `$prog in Node and acyclic[$prog],`
$eQuant$: `some eval ...,`
$aQuant$: `all eval ...)`

1. candidate search

- `solve` $conj \wedge eQuant$

$\rightarrow$ *candidate instance* `$cand`: values of all relations except $eQuant.var$

2. verification

- `solve` $\neg aQuant$ against the `$cand` *partial instance*

$\rightarrow$ *counterexample* `$cex`: value of the $eQuant.var$ relation

partial instance
- partial solution known upfront
- enforced using *bounds*

ALLOY* Implementation **Caveats**

```
some prog: Node |
  acyclic[prog]
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

$\rightarrow$

$\exists\forall(conj$: `$prog in Node and acyclic[$prog],`
$eQuant$: `some eval ...,`
$aQuant$: `all eval ...)`

1. candidate search

- `solve` $conj \wedge eQuant$

$\rightarrow$ *candidate instance* `$cand`: values of all relations except $eQuant.var$

2. verification

- `solve` $\neg aQuant$ against the `$cand` *partial instance*

$\rightarrow$ *counterexample* `$cex`: value of the $eQuant.var$ relation

partial instance
- partial solution known upfront
- enforced using *bounds*

3. induction

- use *incremental solving* to add

  replace $eQuant.var$ **with** `$cex` **in** $eQuant.body$

  to previous search condition

ALLOY* Implementation **Caveats**

```
some prog: Node |
  acyclic[prog]
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

→

$\exists\forall($*conj*: `$prog in Node and acyclic[$prog],`
eQuant: `some eval ...,`
aQuant: `all eval ...)`

1. candidate search

- `solve` *conj* $\wedge$ *eQuant*

→ *candidate instance* `$cand`: values of all relations except *eQuant*.*var*

2. verification

- `solve` $\neg$*aQuant* against the `$cand` *partial instance*

→ *counterexample* `$cex`: value of the *eQuant*.*var* relation

partial instance
- partial solution known upfront
- enforced using *bounds*

3. induction

- use *incremental solving* to add
  `replace` *eQuant*.*var* **`with`** `$cex` **`in`** *eQuant*.*body*
  to previous search condition

incremental solving
- continue from prev solver instance
- the solver reuses learned clauses

ALLOY* Implementation **Caveats**

```
some prog: Node |
  acyclic[prog]
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

$\rightarrow$

$\exists\forall($*conj*: `$prog in Node and acyclic[$prog],`
eQuant: `some eval ...,`
aQuant: `all eval ...`$)$

1. candidate search

- `solve` *conj* $\wedge$ *eQuant*

$\rightarrow$ *candidate instance* `$cand`: values of all relations except *eQuant.var*

2. verification

- `solve` $\neg$*aQuant* against the `$cand` *partial instance*

$\rightarrow$ *counterexample* `$cex`: value of the *eQuant.var* relation

partial instance
- partial solution known upfront
- enforced using *bounds*

3. induction

- use *incremental solving* to add
  replace *eQuant.var* **with** `$cex` **in** *eQuant.body*
  to previous search condition

incremental solving
- continue from prev solver instance
- the solver reuses learned clauses

? *what if the increment formula is not first-order*

– optimization 1: use its weaker "first-order version"

ALLOY* **Optimization**

2. domain constraints

"for all possible `eval`, if the semantics hold then the spec must hold"

vs.

"for all `eval` that satisfy the semantics, the spec must hold"

ALLOY* **Optimization**

2. domain constraints

"for all possible `eval`, if the semantics hold then the spec must hold"	vs.	*"for all `eval` that satisfy the semantics, the spec must hold"*

- logically equivalent, **but**, when "for" implemented as CEGIS:

ALLOY* **Optimization**

2. domain constraints

"for all possible `eval`, if the semantics hold then the spec must hold" vs. *"for all `eval` that satisfy the semantics, the spec must hold"*

- logically equivalent, **but**, when "for" implemented as CEGIS:

```
pred synth[prog: Node] {
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
}
```

↓
candidate search

```
some prog: Node |
  some eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

↓

a valid candidate doesn't have to satisfy the `semantics` predicate!

✗

ALLOY* Optimization

2. domain constraints

“for all possible `eval`, if the semantics hold then the spec must hold” vs. *“for all `eval` that satisfy the semantics, the spec must hold”*

- logically equivalent, **but**, when “for” implemented as CEGIS:

```
pred synth[prog: Node] {
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
}
```

↓

candidate search

```
some prog: Node |
  some eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[prog, eval]
```

↓

a valid candidate doesn’t have to satisfy the `semantics` predicate!

✗

```
pred synth[prog: Node] {
  all eval: Node -> (Int+Bool) when semantics[eval]
    spec[prog, eval]
}
```

↓

candidate search

```
some prog: Node |
  some eval: Node -> (Int+Bool) when semantics[eval] |
    spec[prog, eval]
```

↓

a valid candidate must satisfy the `semantics` predicate!

✔

ALLOY* Evaluation

evaluation goals

evaluation goals

1. scalability on classical higher-order graph problems
   - ? does ALLOY* scale beyond “toy-sized” graphs

ALLOY* Evaluation

evaluation goals

1. scalability on classical higher-order graph problems
   - ? does ALLOY* scale beyond "toy-sized" graphs

2. applicability to program synthesis
   - ? expressiveness: how many SyGuS benchmarks can be written in ALLOY*
   - ? power: how many SyGuS benchmarks can be solved with ALLOY*
   - ? scalability: how does ALLOY* compare to other synthesizers

Alloy* Evaluation

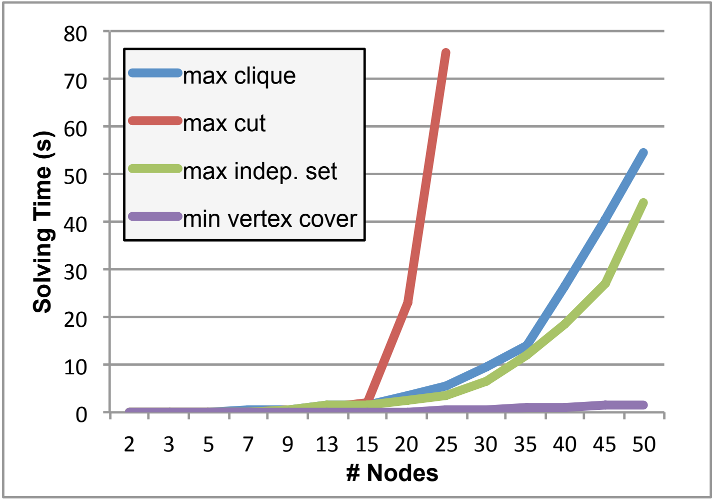
evaluation goals

1. scalability on classical higher-order graph problems
   - ? does Alloy* scale beyond "toy-sized" graphs

2. applicability to program synthesis
   - ? expressiveness: how many SyGuS benchmarks can be written in Alloy*
   - ? power: how many SyGuS benchmarks can be solved with Alloy*
   - ? scalability: how does Alloy* compare to other synthesizers

3. benefits of the two optimizations
   - ? do Alloy* optimizations improve overall solving times

Evaluation: **Graph** Algorithms



Evaluation: Program **Synthesis**

expressiveness

- we extended Alloy to support bit vectors
- we encoded **123/173** benchmarks, i.e., all except "ICFP problems"
  - reason for skipping ICFP: 64-bit bit vectors (not supported by Kodkod)
  - (aside) not one of them was solved by any of the competition solvers

power

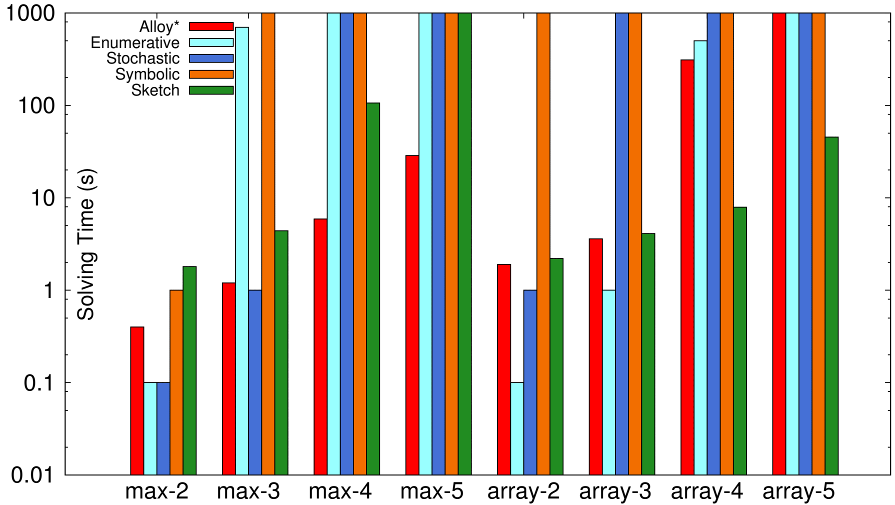
- ALLOY* was able to solve **all** different **categories** of benchmarks
  - integer benchmarks, bit vector benchmarks, let constructs, synthesizing multiple functions at once, multiple applications of the synthesized function

scalability

- many of the 123 benchmarks are either too easy or too difficult
  → not suitable for scalability comparison
- we primarily used the integer benchmarks
- we also picked a few bit vector benchmarks that were too hard for all solvers

Evaluation: Program **Synthesis**

scalability comparison (integer benchmarks)



Evaluation: Program **Synthesis**

scalability comparison (select bit vector benchmarks)

- benchmarks
  - `parity-AIG-d1`: full parity circuit using AND and NOT gates
  - `parity-NAND-d1`: full parity circuit using AND always followed by NOT

Evaluation: Program **Synthesis**

scalability comparison (select bit vector benchmarks)

- benchmarks
  - `parity-AIG-d1`: full parity circuit using AND and NOT gates
  - `parity-NAND-d1`: full parity circuit using AND always followed by NOT
- all solvers (including ALLOY*) time out on both (limit: 1000s)

Evaluation: Program **Synthesis**

scalability comparison (select bit vector benchmarks)

- benchmarks
  - `parity-AIG-d1`: full parity circuit using AND and NOT gates
  - `parity-NAND-d1`: full parity circuit using AND always followed by NOT
- all solvers (including ALLOY*) time out on both (limit: 1000s)
- custom tweaks in ALLOY* synthesis models:
  - create and use a single type of gate
  - impose partial ordering between gates

Evaluation: Program **Synthesis**

scalability comparison (select bit vector benchmarks)

- benchmarks
  - `parity-AIG-d1`: full parity circuit using AND and NOT gates
  - `parity-NAND-d1`: full parity circuit using AND always followed by NOT
- all solvers (including ALLOY*) time out on both (limit: 1000s)
- custom tweaks in ALLOY* synthesis models:
  - create and use a single type of gate
  - impose partial ordering between gates

`parity-AIG-d1`

```
sig AIG extends BoolNode {
  left, right: one BoolNode
  invLhs, invRhs, invOut: one Bool
}
pred aig_semantics[eval: Node->(Int+Bool)] {
  all n: AIG |
    eval[n] = ((eval[n.left] ^ n.invLhs) &&
               (eval[n.right] ^ n.invRhs)
              ) ^ n.invOut}
run synth for 0 but -1..0 Int, exactly 15 AIG
```

`parity-NAND-d1`

```
sig NAND extends BoolNode {
  left, right: one BoolNode
}

pred nand_semantics[eval: Node->(Int+Bool)] {
  all n: NAND |
    eval[n] = !(eval[n.left] &&
                eval[n.right])
}
run synth for 0 but -1..0 Int, exactly 23 NAND
```

Evaluation: Program **Synthesis**

scalability comparison (select bit vector benchmarks)

- benchmarks
  - `parity-AIG-d1`: full parity circuit using AND and NOT gates
  - `parity-NAND-d1`: full parity circuit using AND always followed by NOT
- all solvers (including ALLOY*) time out on both (limit: 1000s)
- custom tweaks in ALLOY* synthesis models:
  - create and use a single type of gate
  - impose partial ordering between gates

parity-AIG-d1

```
sig AIG extends BoolNode {
  left, right: one BoolNode
  invLhs, invRhs, invOut: one Bool
}
pred aig_semantics[eval: Node->(Int+Bool)] {
  all n: AIG |
    eval[n] = ((eval[n.left] ^ n.invLhs) &&
               (eval[n.right] ^ n.invRhs)
              ) ^ n.invOut}
run synth for 0 but -1..0 Int, exactly 15 AIG
```

solving time w/ partial ordering: 20s
solving time w/o partial ordering: 80s

parity-NAND-d1

```
sig NAND extends BoolNode {
  left, right: one BoolNode
}

pred nand_semantics[eval: Node->(Int+Bool)] {
  all n: NAND |
    eval[n] = !(eval[n.left] &&
                eval[n.right])
}
run synth for 0 but -1..0 Int, exactly 23 NAND
```

solving time w/ partial ordering: 30s
solving time w/o partial ordering: ∞

Evaluation: Benefits of ALLOY* **Optimizations**

	base	**w/ optimizations**
`max2`	0.4s	0.3s
`max3`	7.6s	0.9s
`max4`	t/o	1.5s
`max5`	t/o	4.2s
`max6`	t/o	16.3s
`max7`	t/o	163.6s
`max8`	t/o	987.3s
`array-search2`	140.0s	1.6s
`array-search3`	t/o	4.0s
`array-search4`	t/o	16.1s
`array-search5`	t/o	485.6s

	base	**w/ optimizations**
`turan5`	3.5s	0.5s
`turan6`	12.8s	2.1s
`turan7`	235.0s	3.8s
`turan8`	t/o	15.0s
`turan9`	t/o	45.0s
`turan10`	t/o	168.0s

ALLOY* Conclusion

ALLOY* **is**

- general purpose constraint solver
- capable of efficiently solving arbitrary higher-order formulas
- sound & complete within given bounds

ALLOY* Conclusion

ALLOY* is

- general purpose constraint solver
- capable of efficiently solving arbitrary higher-order formulas
- sound & complete within given bounds

higher-order and alloy historically

- bit-blasting higher-order quantifiers: attempted, deemed intractable
- previously many ad hoc mods to alloy
  - aluminum, razor, staged execution, ...

ALLOY* Conclusion

ALLOY* is

- general purpose constraint solver
- capable of efficiently solving arbitrary higher-order formulas
- sound & complete within given bounds

higher-order and alloy historically

- bit-blasting higher-order quantifiers: attempted, deemed intractable
- previously many ad hoc mods to alloy
  - aluminum, razor, staged execution, ...

why is this important?

- accessible to wider audience, encourages new applications
- potential impact
  - abundance of tools that build on Alloy/Kodkod, for testing, program analysis, security, bounded verification, executable specifications, ...

SUNNY: Model-Based Reactive Web Framework

(my previous work)

executable specs for java

Squander

program synthesis

Jennisys

spec formal logic
engine sophisticated search
apps complex algorithms, constraint solving

ARbY [ABZ'14]

• unified specification & implementation language

spec DSL
engine translation/compilation
apps domain-specific uses

ALLOY* [ABZ'12,SCP'14,ICSE'15]

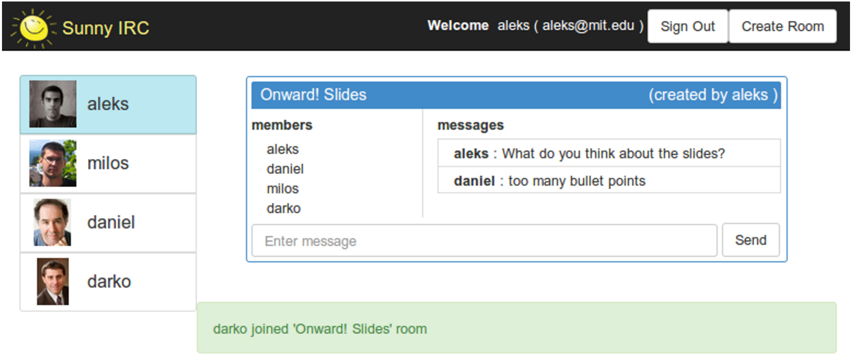
• more powerful constraint solver
• capable of solving a whole new category of formal specifications

SUNNY [Onward'13]

• model-based web framework
• reactive, single-tier, policy-agnostic
• what instead of how

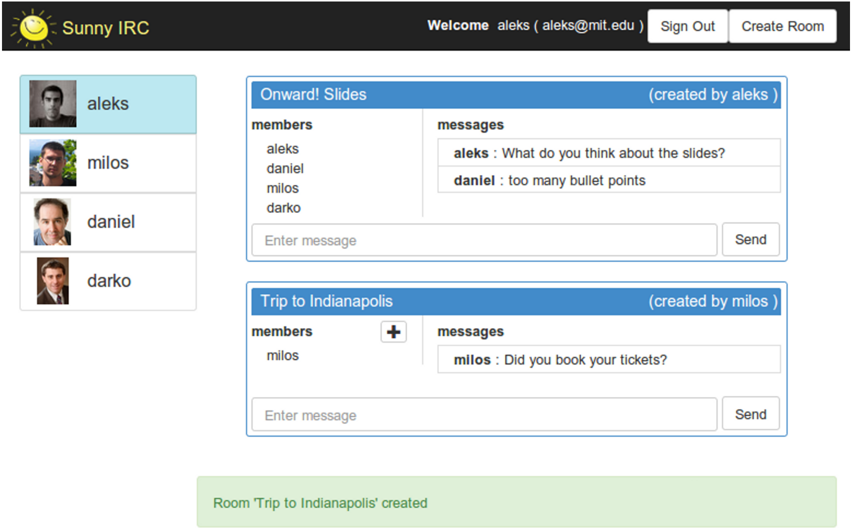
A simple web app: SUNNY IRC

custom-tailored internet chat relay app



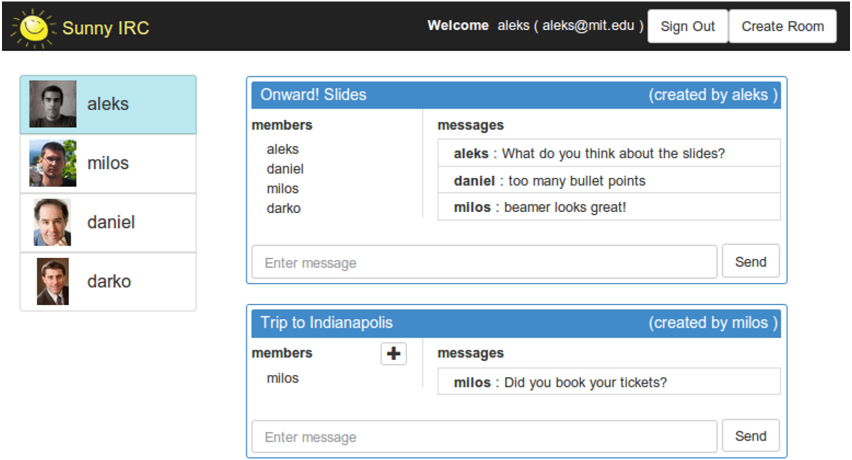
A simple web app: SUNNY IRC

custom-tailored internet chat relay app



A simple web app: SUNNY **IRC**

custom-tailored internet chat relay app



Conceptually **simple, but** in practice...

Conceptually **simple, but** in practice...

- **distributed system**
  - concurrency issues
  - keeping everyone updated

Conceptually **simple, but** in practice...

- **distributed system**
    - concurrency issues
    - keeping everyone updated
- **heterogeneous environment**
    - rails + javascript + ajax + jquery + ...
    - html + erb + css + sass + scss + bootstrap + ...
    - db + schema + server config + routes + ...

Conceptually **simple, but** in practice...

- **distributed system**
  - concurrency issues
  - keeping everyone updated
- **heterogeneous environment**
  - rails + javascript + ajax + jquery + ...
  - html + erb + css + sass + scss + bootstrap + ...
  - db + schema + server config + routes + ...
- **abstraction gap**
  - high-level problem domain
  - low-level implementation level

Conceptually **simple, but** in practice...

- **distributed system**
  - concurrency issues
  - keeping everyone updated

- **heterogeneous environment**
  - rails + javascript + ajax + jquery + ...
  - html + erb + css + sass + scss + bootstrap + ...
  - db + schema + server config + routes + ...

- **abstraction gap**
  - high-level problem domain
  - low-level implementation level

exercise:

sketch out a model (design, spec)
for the Sunny IRC application

Sunny IRC: **data model**

```
user class User
# inherited: name, email: Text

 salute: ()-> "Hi #{this.name}"
```

```
record class Msg
 text: Text
 sender: User
 time: Val
```

```
record class ChatRoom
 name: Text
 members: set User
 messages: compose set Msg
```

- record: automatically persisted objects with typed fields
- user: special kind of record, assumes certain fields, auth, etc.
- set: denotes non-scalar (set) type
- compose: denotes ownership, deletion propagation, etc.

Sunny IRC: **machine model**

```
client class Client
  user: User
```

```
server class Server
  rooms: compose set ChatRoom
```

- client: special kind of record, used to represent client machines
- server: special kind of record, used to represent the server machine

Sunny IRC: **event model**

```
event class SendMsg
 from: client: Client
 to:   server: Server

 params:
  room: ChatRoom
  msgText: Text

 requires: () ->
  return "must log in!"    unless this.client?.user
  return "must join room!" unless this.room?.members.contains(this.client.user)

 ensures: () ->
  this.room.messages.push Msg.create(sender: this.client.user
                                     text: this.msgText
                                     time: Date.now())
```

- to, from: sender and receiver machines
- params: event parameters
- requires: event precondition
- ensures: event handler (postcondition)

challenge

how to make the most of this model?

challenge

how to make the most of this model?

goal

make the model executable as much as possible!

Traditional **MVC** Approach

Traditional **MVC** Approach

- boilerplate:
  - write a matching DB schema
  - turn each record into a resource (model class)
  - turn each event into a controller and implement the CRUD operations
  - configure URL routes for each resource

Traditional **MVC** Approach

- boilerplate:
  - write a matching DB schema
  - turn each record into a resource (model class)
  - turn each event into a controller and implement the CRUD operations
  - configure URL routes for each resource
- aesthetics:
  - design and implement a nice looking HTML/CSS presentation

Traditional **MVC** Approach

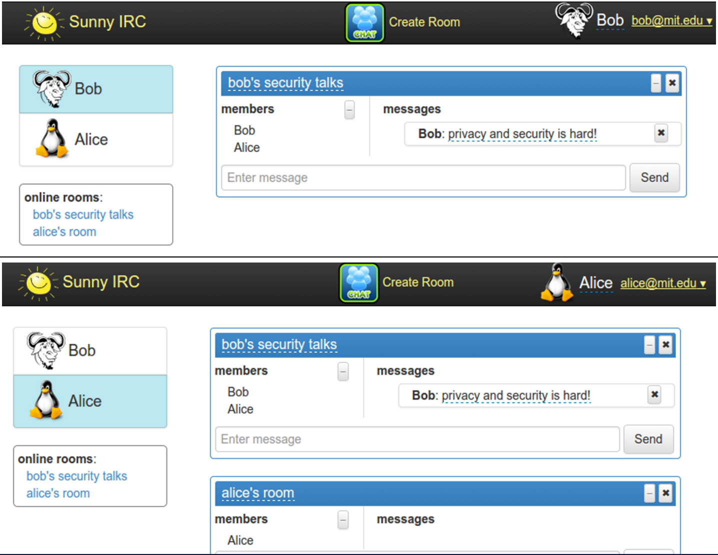
- boilerplate:
  - write a matching DB schema
  - turn each record into a resource (model class)
  - turn each event into a controller and implement the CRUD operations
  - configure URL routes for each resource
- aesthetics:
  - design and implement a nice looking HTML/CSS presentation
- to make it interactive:
  - decide how to implement server push
  - keep track of who's viewing what
  - monitor resource accesses
  - push changes to clients when resources are modified
  - implement client-side Javascript to accept pushed changes and dynamically update the DOM

Traditional **MVC** Approach

- boilerplate:
  - write a matching DB schema
  - turn each record into a resource (model class)
  - turn each event into a controller and implement the CRUD operations
  - configure URL routes for each resource
- aesthetics:
  - design and implement a nice looking HTML/CSS presentation
- to make it interactive:
  - decide how to implement server push
  - keep track of who's viewing what
  - monitor resource accesses
  - push changes to clients when resources are modified
  - implement client-side Javascript to accept pushed changes and dynamically update the DOM

SUNNY demo

demo: responsive GUI without messing with javascript



GUIs in SUNNY: **dynamic templates**

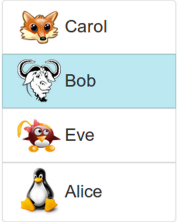
- like standard templating engine with data bindings
- automatically re-rendered when the model changes

GUIs in SUNNY: dynamic templates

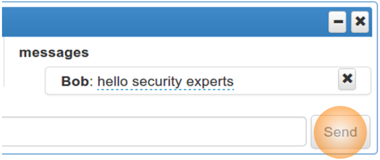
- like standard templating engine with data bindings
- automatically re-rendered when the model changes

online_users.html

```
<div>
  {{#each Server.onlineClients.user}}
    {{> user_tpl user=this}}
  {{/each}}
</div>
```



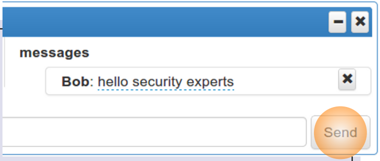
GUIs in SUNNY: **binding to events**



GUIs in SUNNY: binding to events

room_tpl.html

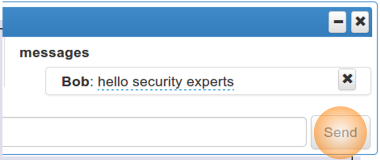
```
<div {{SendMsg room=this.room}} >
  <div>
    <input type="text" name="text"
       placeholder="Enter message"
       {{SendMsg_msgText}}
       {{sunny_trigger}} />
  </div>
  <button {{sunny_trigger}}>Send</button>
</div>
```



GUIs in SUNNY: **binding to events**

room_tpl.html

```
<div {{SendMsg room=this.room}} >
  <div>
    <input type="text" name="text"
       placeholder="Enter message"
       {{SendMsg_msgText}}
       {{sunny_trigger}} />
  </div>
  <button {{sunny_trigger}}>Send</button>
</div>
```



- html5 `data` attributes specify event type and parameters
- dynamically discovered and triggered asynchronously
- no need for any Ajax requests/responses
  - the data-binding mechanism will automatically kick in

Adding New Features: **adding a field**

implement user status messages

Adding New Features: adding a field

implement user status messages

- all it takes:

```
user class User
  status: Text
```

```
<p {{editableField obj=this.user fld="status"}}>
  {{this.user.status}}
</p>
```

Adding New Features: adding a field

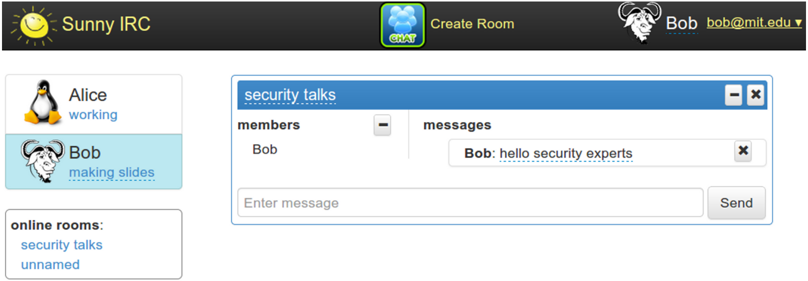
implement user status messages

- all it takes:

```
user class User
  status: Text
```

```
<p {{editableField obj=this.user fld="status"}}>
  {{this.user.status}}
</p>
```

demo



Security/Privacy: **write** policies

forbid changing other people's data

- by default, all fields are public
- policies used to specify access restrictions

Security/Privacy: write policies

forbid changing other people's data

- by default, all fields are public
- policies used to specify access restrictions

```
policy User,
  update:
    "*": (usr, val) ->
      return this.allow() if usr.equals(this.client?.user)
      return this.deny("can't edit other people's data")
```

Security/Privacy: **write** policies

forbid changing other people's data

- by default, all fields are public
- policies used to specify access restrictions

```
policy User,
  update:
    "*": (usr, val) ->
      return this.allow() if usr.equals(this.client?.user)
      return this.deny("can’t edit other people’s data")
```

- declarative and independent from the rest of the system
- automatically checked by the system at each field access

Security/Privacy: **read** & **find** policies

hide avatars unless the two users share a room

Security/Privacy: **read** & **find** policies

hide avatars unless the two users share a room

```
policy User,
 read:
   avatar: (usr) ->
     clntUser = this.client?.user
     return this.allow() if usr.equals(clntUser)
     if (this.server.rooms.some (room)->room.members.containsAll([usr, clntUser]))
       return this.allow()
     else
       return this.deny()
```

- **read denied** → empty value returned instead of raising exception

Security/Privacy: **read** & **find** policies

hide avatars unless the two users share a room

```
policy User,
 read:
   avatar: (usr) ->
     clntUser = this.client?.user
     return this.allow() if usr.equals(clntUser)
     if (this.server.rooms.some (room)->room.members.containsAll([usr, clntUser]))
       return this.allow()
     else
       return this.deny()
```

- **read denied** → empty value returned instead of raising exception

invisible users: hide users whose status is “busy”

Security/Privacy: read & find policies

hide avatars unless the two users share a room

```
policy User,
 read:
   avatar: (usr) ->
     clntUser = this.client?.user
     return this.allow() if usr.equals(clntUser)
     if (this.server.rooms.some (room)->room.members.containsAll([usr, clntUser]))
       return this.allow()
     else
       return this.deny()
```

- **read denied** → empty value returned instead of raising exception

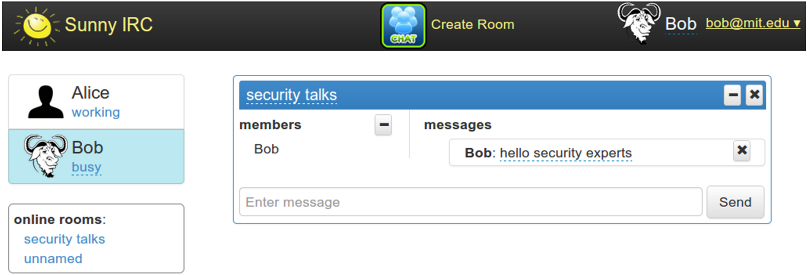
invisible users: hide users whose status is “busy”

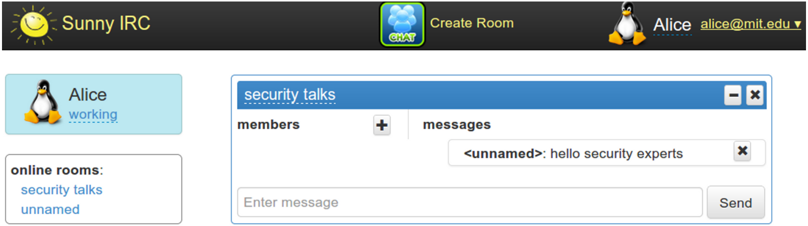
```
policy User,
 find: (users) -> clntUser = this.client?.user
                  return this.allow(filter users, (u) -> u.equals(clntUser) ||
                                                         u.status != "busy")
```

- **find policies** → objects entirely removed from the client-view of the data

Demo: defining **access policies** independently

no GUI templates need to change!





Policy Checking in SUNNY

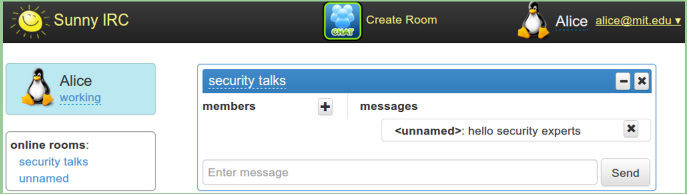
access control style

- policies attached to fields
- implicit principal: client which issued current request
- evaluate against the dynamic state of the program
- policy code executes in the current client context
  - circular dependencies resolved by allowing recursive operations

Policy Checking in SUNNY

access control style

- policies attached to fields
- implicit principal: client which issued current request
- evaluate against the dynamic state of the program
- policy code executes in the current client context
  - circular dependencies resolved by allowing recursive operations
- policy execution creates reactive server-side dependencies



  - Alice's client doesn't contain Bob's status field at all
  - nevertheless, it automatically reacts when Bob changes his status!

Related Work: Reactive + Policies

	checking policies	enforcing policies	reactive

Related Work: Reactive + Policies

	checking policies	enforcing policies	reactive
UI Frameworks (.NET, XAML, Backbone.js, AngularJS, ...)	✗	✗	✔

Related Work: Reactive + Policies

	checking policies	enforcing policies	reactive
UI Frameworks (.NET, XAML, Backbone.js, AngularJS, ...)	✗	✗	✔
Traditional IF (Resin, Jiff, Dytan, ...)	✔	✗	✗

Related Work: Reactive + Policies

	checking policies	enforcing policies	reactive
UI Frameworks (.NET, XAML, Backbone.js, AngularJS, ...)	✗	✗	✔
Traditional IF (Resin, Jiff, Dytan, ...)	✔	✗	✗
Reactive Web (Ur/Web, Elm, Flapjax, Meteor, ...)	✔	✗	✔

Related Work: Reactive + Policies

	checking policies	enforcing policies	reactive
UI Frameworks (.NET, XAML, Backbone.js, AngularJS, ...)	✗	✗	✔
Traditional IF (Resin, Jiff, Dytan, ...)	✔	✗	✗
Reactive Web (Ur/Web, Elm, Flapjax, Meteor, ...)	✔	✗	✔
Enforcing Policies (Jeeves, Hails/LIO, ...)	✔	✔	✗

Related Work: Reactive + Policies

	checking policies	enforcing policies	reactive
UI Frameworks (.NET, XAML, Backbone.js, AngularJS, ...)	✗	✗	✔
Traditional IF (Resin, Jiff, Dytan, ...)	✔	✗	✗
Reactive Web (Ur/Web, Elm, Flapjax, Meteor, ...)	✔	✗	✔
Enforcing Policies (Jeeves, Hails/LIO, ...)	✔	✔	✗
Sunny	✔	✔	✔

Example SUNNY **Apps**

gallery of applications

- internet relay chat
  - implement invisible users with policies
- party planner
  - intricate and interdependent policies for hiding sensitive data
- social network
  - highly customizable privacy settings
- photo sharing
  - similar to “social network”, but in the context of file sharing
- mvc todo
  - from single- to multi-user with policies

SUNNY: the big picture

SUNNY: the big picture

declarative nature of SUNNY

- centralized unified model
- single-tier
- uncluttered focus on essentials: **what** the app should do

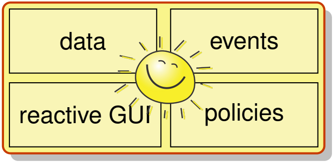
SUNNY: the big picture

declarative nature of SUNNY

- centralized unified model
- single-tier
- uncluttered focus on essentials: **what** the app should do

my contribution: functionality

- separation of main concerns: data, events, GUI, policies



SUNNY: the big picture

declarative nature of SUNNY

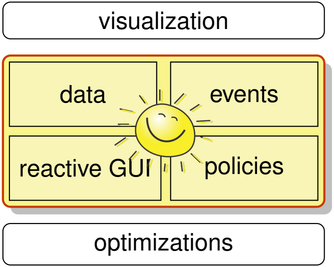
- centralized unified model
- single-tier
- uncluttered focus on essentials: **what** the app should do

my contribution: functionality

- separation of main concerns: data, events, GUI, policies

going forward:

- optimizations
  - scalable/parallelizable back ends
  - clever data partitioning
  - declarative model-based cloud apps
- visualization
  - flexible model-based GUI builder
  - generic & reusable widgets



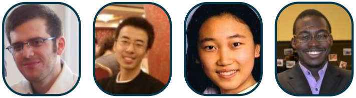
Acknowledgements

Acknowledgements

advisor

thesis committee

UROPs



co-authors/ collaborators

SUNNY: the big picture

declarative nature of SUNNY

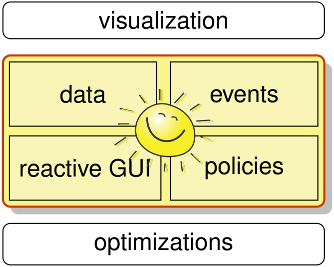
- centralized unified model
- single-tier
- uncluttered focus on essentials: **what** the app should do

my contribution: functionality

- separation of main concerns: data, events, GUI, policies

going forward:

- optimizations
  - scalable/parallelizable back ends
  - clever data partitioning
  - declarative model-based cloud apps
- visualization
  - flexible model-based GUI builder
  - generic & reusable widgets



Thank You!

document