Analyzing Alloy Constraints using an SMT Solver: A Case Study

Aboubakr Achraf El Ghazi and Mana Taghdiri

Karlsruhe Institute of Technology, Germany

July 12, 2010

Motivation

- Checking structure-intensive systems.
- Full automation.
- Existing approaches:
  - Theorem proving
    $\Rightarrow$ sacrifices the full automation
  - Bounded verification
    $\Rightarrow$ sacrifices soundness

Goal:
Full automation but sacrifice soundness just on demand.

Alloy

- A first order relational logic with transitive closure.
- Key ideas:
  - Everything is a relation.
  - Look for counterexamples in a bounded scope.
- Very widely used:
  - Stand-alone constraint solver: file systems, network protocols, ...
  - Back-end engine for program analysis tools: Forge, Karun, Jalloy, ...
- The Alloy analyzer is an *instance finder*.
- For a model M, an assertion A, and a scope S, the Alloy analyzer looks for an instance within S, so that $\neg A$ holds in M.
- A satisfiability problem (SAT):
  - Translates M and $\neg A$ to an equivalent propositional formula using S.
  - If a solution is found, Alloy reports it to the user as a counterexample.
  - If not, then the assertion A is valid in the model, but only w.r.t. S

The Alloy Analyzer

- Benefits:
  - Fully automatic.
  - Designed to be quick –small scope hypothesis.
  - Expressive logic.
  - Exhaustive w.r.t. scope (bounded verification).
- Shortcomings:
  - Can never prove an assertion, even for the simplest models.
  - Increasing confidence by increasing scope doesn't scale well for complex models.
  - Limited support for arrays and numerical constraints.
- Our goal:
  - Solve the above problems while keeping the benefits.

SMT-Solver: Yices

- Checks satisfiability problem modulo a set of theories
- Supports a rich set of theories: uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, dependent types, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and λ-expression.
- Some are decidable some are not.

Approach: Solving Alloy formulas using Yices

- Advantages:
  - (Full) support of (linear) arithmetic, array, ...
    $\Rightarrow$ scalability
  - Type finitization on demand
    $\Rightarrow$ Can prove assertions
    $\Rightarrow$ Increase the confidence
- Challenging: Alloy constructs:
  - Type system.
  - Universal quantifier.
  - Transitive closure.
  - Sequences.
  - Set cardinality.
  - Relational inverse.

Translation: Details by example

```
1:   abstract sig Target {}
2:   sig Addr extends Target {}
3:   abstract sig Name extends Target {}
4:   sig Alias, Group extends Name {}
5:   sig Book {
6:     names: set Name,
7:     addr: names -> some Target
     }{
8:       all a:Alias | lone a.addr)
     }

     pred add (b, b': Book, n: Name, t: Target) {
9:     b'.addr = b.addr + n->t
     }
     pred del (b, b': Book, n: Name, t: Target) {
10:    b'.addr = b.addr - n->t
     }
     fun lookup (b: Book, n: Name): set Addr {
11:    n.^(b.addr) & Addr
     }
     assert delUndoesAdd {
12:    all b, b', b'': Book , n: Name, t: Target |
13:      no n.(b.addr) and
         (
           (
14:          add[b, b', n, t] and
15:          del[b', b'', n, t]
           ) implies
16:        b.addr = b''.addr
         )
     }
 check delUndoesAdd for 4
```

Translation: Details by example

Alloy Model

```
1:  abstract  sig Target {}
2:  sig Addr  extends Target {}
3:  abstract sig Name  extends Target {}
4:  sig Alias, Group  extends Name {}
```

Yices Model

```
1:   (define-type Target)
2:   (define isAddr::(-> Target bool))
2:   (define-type Addr (subtype (t::Target) (isAddr t)))
3:   (define isName::(-> Target bool))
3:   (define-type Name (subtype (t::Target) (isName t)))
2,3: (assert (forall (t::Target)
2,3:  (not (and (isAddr t) (isName t)))
2,3: ))
2,3: (assert (forall (t::Target)
2.3:  (or (isAddr t) (isName t))
2.3: ))
4:   (define isAlias::(-> Name bool))
4:   (define-type Alias (subtype (n::Name) (isAlias n)))
4:   (define isGroup::(-> Name bool))
4:   (define-type Group (subtype (n::Name) (isGroup n)))
4:   (assert (forall (n::Name)
         (not (and (isAlias n) (isGroup n)))
     ))
4:   (assert (forall (n::Name) (or (isAlias n) (isGroup n))
```

Translation: Details by example

Alloy Model

```
1:  abstract  sig Target {}
2:  sig Addr  extends Target {}
3:  abstract sig Name  extends Target {}
4:  sig Alias, Group  extends Name {}
```

Yices Model

```
1:    (define-type Target)
2:    (define isAddr::(-> Target bool))
2:    (define-type Addr (subtype (t::Target) (isAddr t)))
3:    (define isName::(-> Target bool))
3:    (define-type Name (subtype (t::Target) (isName t)))
2,3:  (assert (forall (t::Target)
2,3:   (not (and (isAddr t) (isName t)))
2,3:  ))
2,3:  (assert (forall (t::Target)
2.3:   (or (isAddr t) (isName t))
2.3:  ))
4:    (define isAlias::(-> Name bool))
4:    (define-type Alias (subtype (n::Name) (isAlias n)))
4:    (define isGroup::(-> Name bool))
4:    (define-type Group (subtype (n::Name) (isGroup n)))
4:    (assert (forall (n::Name)
          (not (and (isAlias n) (isGroup n)))
      ))
4:    (assert (forall (n::Name) (or (isAlias n) (isGroup n))
```

Translation: Details by example

Alloy Model

```
1:  abstract  sig Target {}
2:  sig Addr  extends Target {}
3:  abstract sig Name  extends Target {}
4:  sig Alias, Group  extends Name {}
```

Yices Model

```
1:    (define-type Target)
2:    (define isAddr::(-> Target bool))
2:    (define-type Addr (subtype (t::Target) (isAddr t)))
3:    (define isName::(-> Target bool))
3:    (define-type Name (subtype (t::Target) (isName t)))
2,3:  (assert (forall (t::Target)
2,3:   (not (and (isAddr t) (isName t)))
2,3:  ))
2,3:  (assert (forall (t::Target)
2.3:   (or (isAddr t) (isName t))
2.3:  ))
4:    (define isAlias::(-> Name bool))
4:    (define-type Alias (subtype (n::Name) (isAlias n)))
4:    (define isGroup::(-> Name bool))
4:    (define-type Group (subtype (n::Name) (isGroup n)))
4:    (assert (forall (n::Name)
          (not (and (isAlias n) (isGroup n)))
      ))
4:    (assert (forall (n::Name) (or (isAlias n) (isGroup n))
```

Translation: Details by example

Alloy Model

```
1:  abstract  sig Target {}
2:  sig Addr  extends Target {}
3:  abstract sig Name  extends Target {}
4:  sig Alias, Group  extends Name {}
```

$\forall x \in Target \mid x \in Addr \vee x \in Name$

$\forall x \in Target \mid \neg(x \in Addr \wedge x \in Name)$

Yices Model

```
1:    (define-type Target)
2:    (define isAddr::(-> Target bool))
2:    (define-type Addr (subtype (t::Target) (isAddr t)))
3:    (define isName::(-> Target bool))
3:    (define-type Name (subtype (t::Target) (isName t)))
2,3:  (assert (forall (t::Target)
2,3:   (not (and (isAddr t) (isName t)))
2,3:  ))
2,3:  (assert (forall (t::Target)
2.3:   (or (isAddr t) (isName t))
2.3:  ))
4:    (define isAlias::(-> Name bool))
4:    (define-type Alias (subtype (n::Name) (isAlias n)))
4:    (define isGroup::(-> Name bool))
4:    (define-type Group (subtype (n::Name) (isGroup n)))
4:    (assert (forall (n::Name)
          (not (and (isAlias n) (isGroup n)))
      ))
4:    (assert (forall (n::Name) (or (isAlias n) (isGroup n))
```

Translation: Details by example

Alloy Model

```
1:  abstract  sig Target {}
2:  sig Addr  extends Target {}
3:  abstract sig Name  extends Target {}
4:  sig Alias, Group  extends Name {}
```

Yices Model

```
1:   (define-type Target)
2:   (define isAddr::(-> Target bool))
2:   (define-type Addr (subtype (t::Target) (isAddr t)))
3:   (define isName::(-> Target bool))
3:   (define-type Name (subtype (t::Target) (isName t)))
2,3: (assert (forall (t::Target)
2,3:  (not (and (isAddr t) (isName t)))
2,3: ))
2,3: (assert (forall (t::Target)
2.3:  (or (isAddr t) (isName t))
2.3: ))
4:   (define isAlias::(-> Name bool))
4:   (define-type Alias (subtype (n::Name) (isAlias n)))
4:   (define isGroup::(-> Name bool))
4:   (define-type Group (subtype (n::Name) (isGroup n)))
4:   (assert (forall (n::Name)
         (not (and (isAlias n) (isGroup n)))
     ))
4:   (assert (forall (n::Name) (or (isAlias n) (isGroup n))
```

Translation: Details by example

Alloy Model

```
5:   sig Book {
6:     names:  set Name,
7:     addr: names ->  some Target
      }{
8:      all a:Alias |  lone a.addr
    }
```

Yices Model

```
5: (define-type Book)
6: (define names::(-> Book (-> Name bool)))
7: (define-type addrRange (-> Name (-> Target bool)))
7: (define-type addrType (-> Book (-> Name (-> Target bool)
7: (define choose::addrType)
7: (define oneTarget::(-> Book (-> Name Target)))
7: (define addr::addrType
7:     (lambda (b::Book)
7:         (lambda (n::Name)
7:             (lambda (t::Target)
7:                 (if (not ((names b) n))
7:                     false
7:                     (if (= t ((oneTarget b) n))
7:                         true
8:                         (if (isAlias n)
8:                             false
7:                             (((choose b) n) t)
    )))))))
```

Translation: Details by example

Alloy Model

```
5:   sig Book {
6:     names:  set Name,
7:     addr: names ->  some Target
      }{
8:      all a:Alias |  lone a.addr
    }
```

Yices Model

```
5: (define-type Book)
6: (define names::(-> Book (-> Name bool)))
7: (define-type addrRange (-> Name (-> Target bool)))
7: (define-type addrType (-> Book (-> Name (-> Target bool)
7: (define choose::addrType)
7: (define oneTarget::(-> Book (-> Name Target)))
7: (define addr::addrType
7:     (lambda (b::Book)
7:         (lambda (n::Name)
7:             (lambda (t::Target)
7:                 (if (not ((names b) n))
7:                     false
7:                     (if (= t ((oneTarget b) n))
7:                         true
8:                         (if (isAlias n)
8:                             false
7:                             (((choose b) n) t)
    )))))))
```

Translation: Details by example

Alloy Model

```
5:   sig Book {
6:     names:  set Name,
7:     addr: names ->  some Target
      }{
8:      all a:Alias |  lone a.addr
    }
```

Yices Model

```
5: (define-type Book)
6: (define names::(-> Book (-> Name bool)))
7: (define-type addrRange (-> Name (-> Target bool)))
7: (define-type addrType (-> Book (-> Name (-> Target bool)
7: (define choose::addrType)
7: (define oneTarget::(-> Book (-> Name Target)))
7: (define addr::addrType
7:     (lambda (b::Book)
7:         (lambda (n::Name)
7:             (lambda (t::Target)
7:                 (if (not ((names b) n))
7:                     false
7:                     (if (= t ((oneTarget b) n))
7:                         true
8:                         (if (isAlias n)
8:                             false
7:                             (((choose b) n) t)
    )))))))
```

Translation: Details by example

Alloy Model

```
5:   sig Book {
6:     names:  set Name,
7:     addr: names ->  some Target
      }{
8:      all a:Alias |  lone a.addr
    }
```

Yices Model

```
5: (define-type Book)
6: (define names::(-> Book (-> Name bool)))
7: (define-type addrRange (-> Name (-> Target bool)))
7: (define-type addrType (-> Book (-> Name (-> Target bool)
7: (define choose::addrType)
7: (define oneTarget::(-> Book (-> Name Target)))
7: (define addr::addrType
7:     (lambda (b::Book)
7:         (lambda (n::Name)
7:             (lambda (t::Target)
7:                 (if (not ((names b) n))
7:                     false
7:                     (if (= t ((oneTarget b) n))
7:                         true
8:                         (if (isAlias n)
8:                             false
7:                             (((choose b) n) t)
    )))))))
```

Acyclic address book: Transitive Closure

- Acyclicity:

```
fact{
  all b: Book, n: Name |  not (n in n. ^(b.addr))
}
```

- Translation for a relation $r : A \longrightarrow A$:
  - Bound type A to maximum n elements.
  - Calculate: $\hat{}r = r + r.r + r.r.r + \cdots + r^{(n)}$
  - We need the translation of: *Union*, *Join* and *Iterative join*

Union

- For $f, g : A \rightarrow (B \rightarrow bool)$:
  $((union\ f\ g)\ a\ b) = (f\ a\ b) \vee (g\ a\ b)$
- Yices translation:

```
(define-type relType (-> A (-> B bool)))
(define union::(-> relType relType relType)
 (lambda (f::relType g::relType)
  (lambda (a::A)
   (lambda (b::B)
    (or ((f a) b) ((g a) b))
))))
```

Join

- For binary relations, $f.g$ is a relation of arity 2 defined as: $\{(a, c) \mid \exists b.\ (a, b) \in f \wedge (b, c) \in g\}$
- Yices translation:

```
(define-type relType1::(-> A (-> B bool)))
(define-type relType2::(-> B (-> C bool)))
(define-type relType3::(-> A (-> C bool)))

(define join::(-> relType1 relType2 relType3)
  (lambda (f::relType1 g::relType2)
    (lambda (a::A)
      (lambda (c::C)
        (exists (b::B) (and ((f a) b) ((g b) c)))
))))
```

IterJoin and transitive closure (tc)

- $(iterJoin\ i\ f) = f^{(i)}$:

```
(define-type relType::(-> A (-> A bool)))

(define iterJoin::(-> nat relType relType)
  (lambda (i::nat r::relType)
    (if (= i 1) r (join r (iterJoin (- i 1) r)))
))
```

- $(tc\ i\ f) = r + r.r + \cdots + r^{(i)}$:

```
(define tc::(-> nat relType relType)
  (lambda (i::nat r::relType)
    (if (= i 1) r (union (tc (- i 1) r) (iterJoin i r)))
))
```

Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- *delUndoesAdd*: The delete operation undoes the addition operation.
- *addIdempotent*: Repeating an addition has no effect.
- *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- ✓ *delUndoesAdd*: The delete operation undoes the addition operation.
- *addIdempotent*: Repeating an addition has no effect.
- *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- ✓ *delUndoesAdd*: The delete operation undoes the addition operation.
- ✓ *addIdempotent*: Repeating an addition has no effect.
- *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- ✓ *delUndoesAdd*: The delete operation undoes the addition operation.
- ✓ *addIdempotent*: Repeating an addition has no effect.
- ↯ *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

Evaluation: Alloy vs. Yices

Model	Assertion	Scope	Alloy time (s)	Yices time (s)	Tautology?
Basic	delUndoesAdd	30	45.82	0.0006	Yes
		35	**time-out**	0.0006	
	addIdempotent	40	141.27	0.0006	Yes
		50	**time-out**	0.0006	
	addLocal	60	102.41	0.0003	Yes
		70	**memory-out**	0.0003	
Hierarchical	delUndoesAdd	40	139.25	0.009	Yes
		50	**time-out**	0.009	
	addIdempotent	30	23.71	0.008	Yes
		40	**time-out**	0.008	
	addLocal	$n = 2$	0.19	0.02	No
		$n = 3$	0.13	**time-out**	
Acyclic	delUndoesAdd	$n = 6$	150.31	8.10	Don't know
		$n = 7$	**time-out**	29.45	
	addIdempotent	$n = 6$	135.59	8.10	Don't know
		$n = 7$	**time-out**	30.20	
	addLocal	$n = 2$	0.18	0.07	No
		$n = 3$	0.23	**time-out**	

Evaluation: Discussion

- Our analysis is able to prove validity (no type finitization needed) in 5 out of 9 cases.
- Even finitization is made on-demand.
- Outperform Alloy in all but 2 cases:
  - Proved cases are instantaneous.
  - Some uses of transitive closure, as in *addLocal*, are too difficult.

The *addLocal* assertion

Alloy Model

```
...
    fun lookup (b: Book, n: Name): set Addr{
1:    n.^(b.addr) & Addr
   }

    assert addLocal {
2:    all b, b':Book, n, n':Name, t:Target |
3:      (
          n!=n' and
4:        add[b, b', n, t]
        )
         implies
5:      lookup[b,n'] = lookup[b', n']
   }
```

Yices Model

```
...
1: (define lookup::
1:    (-> nat addrRange Name (-> Target bool))
1:   (lambda (i::nat addr::addrRange n::Name)
1:     (lambda (t::Target)
1:       (and (AddrMember t) (((tc i addr) n) t))
1:     )
1:   )
1: )
...
5: (assert
5:    (/=
5:       ((lookup i (addr b_1) n_2) t_2)
5:       ((lookup i (addr b_2) n_2) t_2)
5:    )
5:  )
```

The *addLocal* assertion

Alloy Model

```
...
    fun lookup (b: Book, n: Name): set Addr{
1:    n.^(b.addr) & Addr
    }

    assert addLocal {
2:    all b, b':Book, n, n':Name, t:Target |
3:      (
          n!=n' and
4:        add[b, b', n, t]
        )
        implies
5:      lookup[b,n'] = lookup[b', n']
    }
```

Yices Model

```
...
1: (define lookup::
1:    (-> nat addrRange Name (-> Target bool))
1:    (lambda (i::nat addr::addrRange n::Name)
1:      (lambda (t::Target)
1:        (and (AddrMember t) (((tc i addr) n) t))
1:      )
1:    )
1: )
...
5: (assert
5:    (/=
5:       ((lookup i (addr b_1) n_2) t_2)
5:       ((lookup i (addr b_2) n_2) t_2)
5:    )
5:  )
```

Discussion and future works

- The first step of a novel analysis technique for Alloy models.
- Done by translation to SMT-language (Yices).
- Finitization done on demand:
  - Only when certain language constructs are encountered.
  - Capability of proving validity.
  - Improving performance and scalability.
- A witness of feasibility.
- We bank on the support of quantifiers and λ-exp in SMT-Solver.
  - Yices instances may be unsound (*unknown*)
  - Strategies to deal with *unknown* instances.
- Better solutions to solve transitive closure constraints.
- Covering the entire Alloy language.

Related Work I


A. Armando, J. Mantovani, and L. Platania.
Bounded model checking of software using SMT solvers instead of SAT solvers.
STTT, 11(1):69–83, 2009.


M. Botincan, M. Parkinson, and W. Schulte.
Separation logic verification of c programs with an SMT solver.
ENTCS, 254:5–23, 2009.


L. Erkök and J. Matthews.
Using yices as an automated solver in Isabelle/HOL.
In *AFM*, 2008.


S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli.
Towards smt model-checking of array-based systems.
In *IJCAR*, 2008.

Related Work II


S. Ghilardi and S. Ranise.
Model checking modulo theory at work: the intergration of yices in MCMT.
In *AFM*, 2009.


R. Leino and R. Monahan.
Reasoning about comprehensions with first-order SMT solvers.
In *SAC*, pages 615–622, 2009.

Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- *delUndoesAdd*: The delete operation undoes the addition operation.
- *addIdempotent*: Repeating an addition has no effect.
- *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- ✓ *delUndoesAdd*: The delete operation undoes the addition operation.
- *addIdempotent*: Repeating an addition has no effect.
- *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- ✓ *delUndoesAdd*: The delete operation undoes the addition operation.
- ✓ *addIdempotent*: Repeating an addition has no effect.
- *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

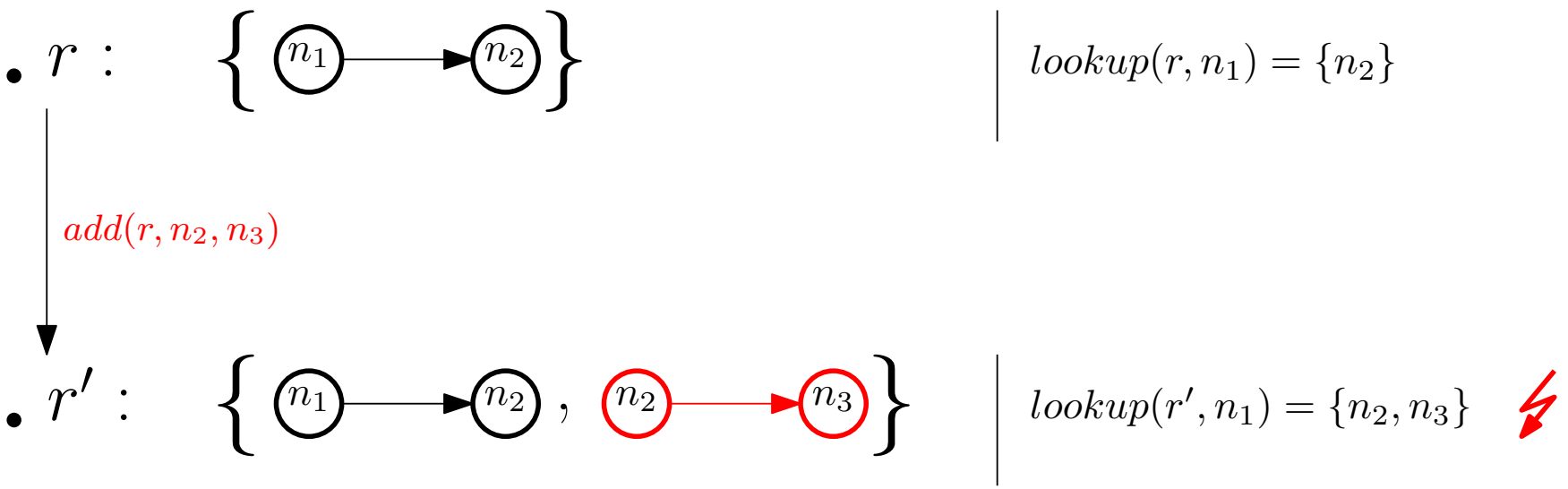
Evaluation

We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- ✓ *delUndoesAdd*: The delete operation undoes the addition operation.
- ✓ *addIdempotent*: Repeating an addition has no effect.
- ↯ *addLocal*: Adding a name n does not affect the addresses reachable from another name n'.

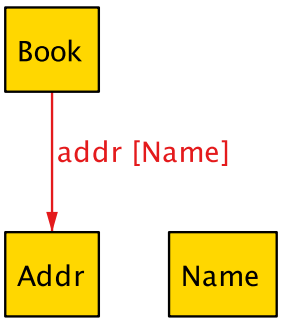
Evaluation

- ↯ *addLocal*: Adding a name *n* does not affect the addresses reachable from another name *n′*.

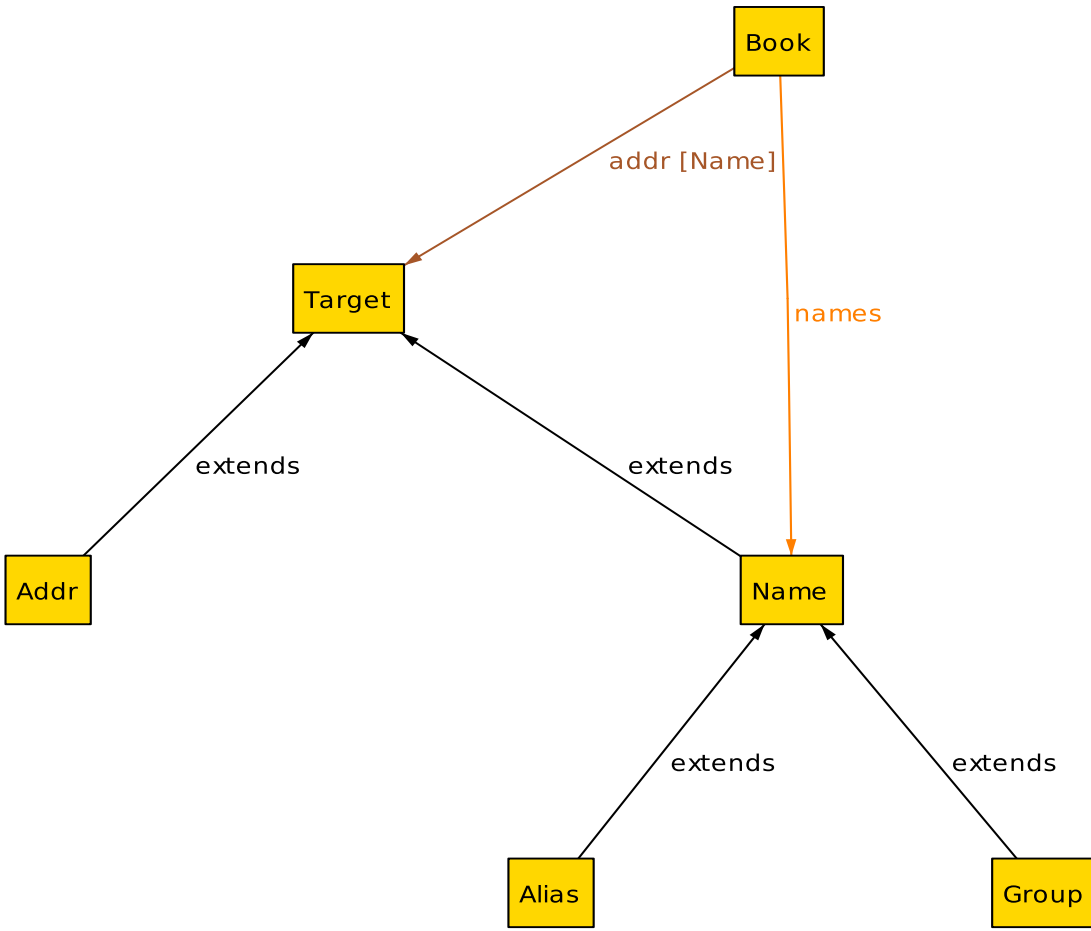


Address book: Type hierarchies

Basic address book:



Hierarchical and acyclic address book:



Hierarchical address Book: Alloy

```
1:   abstract sig Target {}
2:   sig Addr extends Target {}
3:   abstract sig Name extends Target {}
4:   sig Alias, Group extends Name {}
5:   sig Book {
6:     names: set Name,
7:     addr: names -> some Target
     }{
8:       all a:Alias | lone a.addr)
     }

     pred add (b, b': Book, n: Name, t: Target) {
9:     b'.addr = b.addr + n->t
     }
     pred del (b, b': Book, n: Name, t: Target) {
10:    b'.addr = b.addr - n->t
     }
     fun lookup (b: Book, n: Name): set Addr {
11:    n.^(b.addr) & Addr
     }
     assert delUndoesAdd {
12:     all b, b', b'': Book , n: Name, t: Target |
13:       no n.(b.addr) and
14:      add[b, b', n, t] and
15:      del[b', b'', n, t] implies
16:         b.addr = b''.addr
     }
```

Hierarchical address Book: Alloy

```
1:  abstract sig Target {}
2:  sig Addr extends Target {}
3:  abstract sig Name extends Target {}
4:  sig Alias, Group extends Name {}
5:  sig Book {
6:    names: set Name,
7:    addr: names -> some Target
    }{
8:     all a:Alias | lone a.addr)
    }

    pred add (b, b': Book, n: Name, t: Target) {
9:    b'.addr = b.addr + n->t
    }
    pred del (b, b': Book, n: Name, t: Target) {
10:   b'.addr = b.addr - n->t
    }
    fun lookup (b: Book, n: Name): set Addr {
11:   n.^(b.addr) & Addr
    }
    assert delUndoesAdd {
12:    all b, b', b'': Book , n: Name, t: Target |
13:      no n.(b.addr) and
14:     add[b, b', n, t] and
15:     del[b', b'', n, t] implies
16:        b.addr = b''.addr
    }
```

Hierarchical address Book: Alloy

```
1:   abstract sig Target {}
2:   sig Addr extends Target {}
3:   abstract sig Name extends Target {}
4:   sig Alias, Group extends Name {}
5:   sig Book {
6:     names: set Name,
7:     addr: names -> some Target
     }{
8:       all a:Alias | lone a.addr)
     }

     pred add (b, b': Book, n: Name, t: Target) {
9:     b'.addr = b.addr + n->t
     }
     pred del (b, b': Book, n: Name, t: Target) {
10:    b'.addr = b.addr - n->t
     }
     fun lookup (b: Book, n: Name): set Addr {
11:    n.^(b.addr) & Addr
     }
     assert delUndoesAdd {
12:      all b, b', b'': Book , n: Name, t: Target |
13:        no n.(b.addr) and
14:       add[b, b', n, t] and
15:       del[b', b'', n, t] implies
16:          b.addr = b''.addr
     }
```

Hierarchical address Book: Alloy

```
1:  abstract sig Target {}
2:  sig Addr extends Target {}
3:  abstract sig Name extends Target {}
4:  sig Alias, Group extends Name {}
5:  sig Book {
6:    names: set Name,
7:    addr: names -> some Target
    }{
8:     all a:Alias | lone a.addr)
    }

    pred add (b, b': Book, n: Name, t: Target) {
9:    b'.addr = b.addr + n->t
    }
    pred del (b, b': Book, n: Name, t: Target) {
10:   b'.addr = b.addr - n->t
    }
    fun lookup (b: Book, n: Name): set Addr {
11:   n.^(b.addr) & Addr
    }
    assert delUndoesAdd {
12:    all b, b', b'': Book , n: Name, t: Target |
13:      no n.(b.addr) and
14:     add[b, b', n, t] and
15:     del[b', b'', n, t] implies
16:        b.addr = b''.addr
    }
```

$\forall x \in Target \mid x \in Addr \vee x \in Name$

$\forall x \in Target \mid \neg(x \in Addr \wedge x \in Name)$

Hierarchical address Book: Alloy

```
1:   abstract sig Target {}
2:   sig Addr extends Target {}
3:   abstract sig Name extends Target {}
4:   sig Alias, Group extends Name {}
5:   sig Book {
6:     names: set Name,
7:     addr: names -> some Target
     }{
8:       all a:Alias | lone a.addr)
     }

     pred add (b, b': Book, n: Name, t: Target) {
9:     b'.addr = b.addr + n->t
     }
     pred del (b, b': Book, n: Name, t: Target) {
10:    b'.addr = b.addr - n->t
     }
     fun lookup (b: Book, n: Name): set Addr {
11:    n.^(b.addr) & Addr
     }
     assert delUndoesAdd {
12:     all b, b', b'': Book , n: Name, t: Target |
13:       no n.(b.addr) and
14:       add[b, b', n, t] and
15:       del[b', b'', n, t] implies
16:          b.addr = b''.addr
     }
```

Hierarchical address Book: Alloy

```
1:   abstract sig Target {}
2:   sig Addr extends Target {}
3:   abstract sig Name extends Target {}
4:   sig Alias, Group extends Name {}
5:   sig Book {
6:     names: set Name,
7:     addr: names -> some Target
     }{
8:       all a:Alias | lone a.addr)
     }

     pred add (b, b': Book, n: Name, t: Target) {
9:     b'.addr = b.addr + n->t
     }
     pred del (b, b': Book, n: Name, t: Target) {
10:    b'.addr = b.addr - n->t
     }
     fun lookup (b: Book, n: Name): set Addr {
11:    n.^(b.addr) & Addr
     }
     assert delUndoesAdd {
12:     all b, b', b'': Book , n: Name, t: Target |
13:       no n.(b.addr) and
14:       add[b, b', n, t] and
15:       del[b', b'', n, t] implies
16:          b.addr = b''.addr
     }
```

Hierarchical address Book: Alloy

```
1:   abstract  sig Target {}
2:   sig Addr  extends Target {}
3:   abstract sig Name  extends Target {}
4:   sig Alias, Group  extends Name {}
5:   sig Book {
6:     names:  set Name,
7:     addr: names ->  some Target
     }{
8:      all a:Alias |  lone a.addr)
    }

     pred add (b, b': Book, n: Name, t: Target) {
9:     b'.addr = b.addr + n->t
    }
     pred del (b, b': Book, n: Name, t: Target) {
10:    b'.addr = b.addr - n->t
    }
     fun lookup (b: Book, n: Name):  set Addr {
11:    n.^(b.addr)  & Addr
    }
     assert delUndoesAdd {
12:     all b, b', b'': Book , n: Name, t: Target |
13:       no n.(b.addr)  and
14:      add[b, b', n, t]  and
15:      del[b', b'', n, t]  implies
16:         b.addr = b''.addr
    }
```

```
     fact {
8:      all b: Book, a:Alias |  lone a.(b.addr)
    }
```