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 $\lambda$-expression.
- Some are decidable some are not.
Approach: Solving Alloy formulas using Yices

• Advantages:
  • (Full) support of (linear) arithmetic, array, ...
    ⇒ scalability
  • Type finitization on demand
    ⇒ Can prove assertions
    ⇒ 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

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)
4: (not (and (isAlias n) (isGroup n))))
4: ))
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)
4: (not (and (isAlias n) (isGroup n)))
4: ))
4: (assert (forall (n::Name) (or (isAlias n) (isGroup n))))
Translation: Details by example

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) (not (and (isAddr t) (isName t))))
2,3: )
2,3: (assert (forall (t::Target) (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: )
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 {}

∀x ∈ Target | x ∈ Addr ∨ x ∈ Name
∀x ∈ Target | ¬(x ∈ Addr ∧ x ∈ 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)
4: (not (and (isAlias n) (isGroup n))))
4: ))
4: (assert (forall (n::Name) (or (isAlias n) (isGroup n))))
Translation: Details by example

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)
4: (not (and (isAlias n) (isGroup n)))
4: ))
4: (assert (forall (n::Name) (or (isAlias n) (isGroup n)))

Alloy Model

1: abstract sig Target {}
2: sig Addr extends Target {}
3: abstract sig Name extends Target {}
4: sig Alias, Group extends Name {}
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
7:           (if (isAlias n)
7:             false
7:             (((choose b) n) t)
7: ))))))))
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  
7:             (if (isAlias n)  
7:               false  
7:               (((choose b) n) t)  
7:             ))))))))}
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 
8: (lambda (b::Book) 
9: (lambda (n::Name) 
10: (lambda (t::Target) 
11: (if (not ((names b) n)) 
12: false 
13: (if (= t ((oneTarget b) n)) 
14: true 
15: (if (isAlias n) 
16: false 
17: (((choose b) n) t) 
18: ))))))))

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 $r : A \rightarrow A$:
  
  - Bound type $A$ to maximum $n$ elements.
  - Calculate: $\wedge r = r + r.r + r.r.r + \cdots + r^{(n)}$
  - We need the translation of: Union, Join and Iterative join
• For $f, g : A \rightarrow (B \rightarrow bool)$:
  $((union f g) a b) = (f a b) \lor (g a b)$

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

• Yices translation:

```plaintext
(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)

- \((\text{IterJoin } i \ f) = f^{(i)}:\)

  (define-type relType::(\rightarrow A (\rightarrow A \text{ bool})))

  (define IterJoin::(\rightarrow \text{nat} relType relType)
   (lambda (i::nat r::relType)
     (if (= i 1) r (join r (IterJoin (- i 1) r)))
   ))

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

  (define tc::(\rightarrow \text{nat} relType relType)
   (lambda (i::nat r::relType)
     (if (= i 1) r (union (tc (- i 1) r) (IterJoin i r))
   ))
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 \( n' \).
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

<table>
<thead>
<tr>
<th>Model</th>
<th>Assertion</th>
<th>Scope</th>
<th>Alloy time (s)</th>
<th>Yices time (s)</th>
<th>Tautology?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>delUndoesAdd</td>
<td>30</td>
<td>45.82</td>
<td>0.0006</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35</td>
<td>time-out</td>
<td>0.0006</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addIdempotent</td>
<td>40</td>
<td>141.27</td>
<td>0.0006</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>time-out</td>
<td>0.0006</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addLocal</td>
<td>60</td>
<td>102.41</td>
<td>0.0003</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>memory-out</td>
<td>0.0003</td>
<td></td>
</tr>
<tr>
<td>Hierarchical</td>
<td>delUndoesAdd</td>
<td>40</td>
<td>139.25</td>
<td>0.009</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>time-out</td>
<td>0.009</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addIdempotent</td>
<td>30</td>
<td>23.71</td>
<td>0.008</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>time-out</td>
<td>0.008</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addLocal</td>
<td>(n = 2)</td>
<td>0.19</td>
<td>0.02</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 3)</td>
<td>0.13</td>
<td>time-out</td>
<td></td>
</tr>
<tr>
<td>Acyclic</td>
<td>delUndoesAdd</td>
<td>(n = 6)</td>
<td>150.31</td>
<td>8.10</td>
<td>Don’t know</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 7)</td>
<td>time-out</td>
<td>29.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addIdempotent</td>
<td>(n = 6)</td>
<td>135.59</td>
<td>8.10</td>
<td>Don’t know</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 7)</td>
<td>time-out</td>
<td>30.20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addLocal</td>
<td>(n = 2)</td>
<td>0.18</td>
<td>0.07</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n = 3)</td>
<td>0.23</td>
<td>time-out</td>
<td></td>
</tr>
</tbody>
</table>
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 \textit{addLocal}, are too difficult.
The *addLocal* assertion

**Alloy Model**

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

assert addLocal {
  all b, b':Book, n, n':Name, t:Target |
  n!=n' and
  add[b, b', n, t]
} implies
lookup[b,n'] = lookup[b', n']
...```

**Yices Model**

```yices
... (define lookup::
  (-> nat addrRange Name (-> Target bool))
  (lambda (i::nat addr::addrRange n::Name)
    (lambda (t::Target)
      (and (AddrMember t) (((tc i addr) n) t))
    )
  )
... 

(assert
  (/= ((lookup i (addr b_1) n_2) t_2)
    ((lookup i (addr b_2) n_2) t_2)
  )
)```
The *addLocal* assertion

**Alloy Model**

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

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

**Yices Model**

```yices
(define lookup::
  (-> nat addrRange Name (-> Target bool))
  (lambda (i::nat addr::addrRange n::Name)
    (lambda (t::Target)
      (and (AddrMember t) (((tc i addr) n) t))
    )
  )
)

(assert
  (/= ((lookup i (addr b_1) n_2) t_2)
    ((lookup i (addr b_2) n_2) t_2)
  )
)
```
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 \( \lambda \)-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


Related Work II

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

R. Leino and R. Monahan.
Reasoning about comprehensions with first-order SMT solvers.
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 $n'$. 
We have evaluated our analysis by checking the 3 assertions of the 3 variants of the address book model.

- ✓ *delUndoAdd*: 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 \( n' \).
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 $n'$. 
• $\frac{\Box}{\Box} \text{ addLocal}: \text{ Adding a name } n \text{ does not affect the addresses reachable from another name } n'$.

\[
\begin{align*}
\text{r} & : \begin{cases} n_1 \rightarrow n_2 \end{cases} \\
\text{add}(r, n_2, n_3) & \\
\text{r'} & : \begin{cases} n_1 \rightarrow n_2, n_2 \rightarrow n_3 \end{cases} \\
\text{lookup}(r, n_1) &= \{n_2\} \\
\text{lookup}(r', n_1) &= \{n_2, n_3\}
\end{align*}
\]
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)

9: pred add (b, b’: Book, n: Name, t: Target) {
10: b’.addr = b.addr + n->t
}
11: pred del (b, b’: Book, n: Name, t: Target) {
12: b’.addr = b.addr – n->t
}
13: fun lookup (b: Book, n: Name): set Addr {
14: n.(b.addr) & Addr
}
15: assert delUndoesAdd {
16: all b, b’, b’’: Book , n: Name, t: Target | 
17: no n.(b.addr) and
18: add[b, b’, n, t] and
19: del[b’, b’’, n, t] implies
20: b.addr = b’’.addr
}
Hierarchical address Book: Alloy

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

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

```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)
9: 
10:  pred add (b, b': Book, n: Name, t: Target) {
11:      b'.addr = b.addr + n->t
12:  }
13:  pred del (b, b': Book, n: Name, t: Target) {
14:      b'.addr = b.addr - n->t
15:  }
16:  fun lookup (b: Book, n: Name): set Addr {
17:    n.^~(b.addr) & Addr
18:  }
19:  assert delUndoesAdd {
20:    all b, b', b'': Book , n: Name, t: Target |
21:      no n.(b.addr) and
22:      add[b, b', n, t] and
23:      del[b', b'', n, t] implies
24:      b.addr = b''.addr
25:  }
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
}