Engenharia de Software e Sistemas

Uma introdução a Alloy

Márcio Lopes Cornélio
mlc2@cin.ufpe.br

Centro de Informática—UFPE

Um catálogo de endereços

Características

- Desenvolvido por meio de pequenas adições e modificações
- Leve e incremental

Aplicação

- Catálogo de endereços
  - Base de dados que associa endereços com nomes mais curtos que são convenientes para os usuários
- O usuário pode criar um *alias* (um apelido) para um correspondente
- Um grupo semelhante a um *alias*, mas associado a um grupo de correspondentes

Um catálogo de endereços

A jornada

O início

- Um catálogo com *aliases*, mas sem grupos
- Declaração da estrutura do estado de um sistema
- Como gerar amostras de instâncias do estado
- Comportamento dinâmico: como descrever uma operação com restrições, como simular, como verificar propriedades de operações

Evolução

- Estado elaborado para permitir nomes (grupos e *aliases*)
- Estrutura de acordo com o padrão *Composite*
- Traços

Um catálogo de endereços

Modelo

```
sig Name, Addr { }

sig Book {
    addr: Name -> lone Addr
}

pred show { }
```

Um catálogo de endereços

Modelo

Caractersticas

- Assinaturas (Name, Addr, Book) representam conjuntos de objetos
  - Campos, atributos (*fields*)
  - O campo *addr* mapeia nome em endereços. Associa catálogos, nomes e endereços.
  - A expressão b.addr denota o mapeamento de nomes a endereços para o catálogo b
- Multiplicidade
  - A palavra *lone* indica que cada nome é mapeado em, no máximo, um endereço

- O modelo não contém comandos.
  - Limitado à verificação de tipos e semântica estática

Análise

- Predicado com corpo vazio

  pred show ()

- Comando

  run show **for** 3 **but** 1 Book

  especifica o escopo que limita a busca por instâncias

- Resultado: no máximo três objetos por assinatura, exceto Book que pode ter apenas um objeto

- Uma instância com mais de uma associação. Adição de uma restrição ao predicado show

```
pred show[b: Book]{
 #b.addr > 1
}
```

- Verificando se o modelo permite mapear um nome em dois endereços

```
pred show [b: Book] {
  #b.addr > 1
  some n: Name | #n.(b.addr) > 1
}
```

- O analisador aponta uma inconsistência do predicado show

- Porém, é possível haver mais de dois endereços em um catálogo

```
pred show [b: Book] {
    #b.addr > 1
    #Name.(b.addr) > 1
}
```

- A expressão #Name.(b.addr) denota o conjunto de todos os endereços que podem resultar de buscas

- Acrescentando uma operação para adição de endereços

```
sig Book {
    addr: Name -> lone Addr
}

pred add [b, b': Book, n: Name, a: Addr] {
    b'.addr = b.addr + n->a
}
```

- O predicado add é uma restrição e também representa uma operação, descreve comportamento dinâmico

Operação de adição

- Argumentos:
  - um catálogo antes da adição b
  - um catálogo após a adição b'
  - um nome n
  - um endereço a
- Restrições
  - O mapeamento do novo catálogo é igual ao mapeamento do catálogo antigo, adicionando uma nova associação entre nome e endereço

Estilo

- Procedimento em um programa imperativo: operacional, descreve como produzir uma mudança de estado
- Alloy: declarativo, descreve como verificar se uma mudança de estado é válida

Operação de adição

- Executando uma transição de estado: **run** add **for** 3 **but** 2 Book
- Transição com restrições (estado final possui mais de endereço mapeado)

```
pred showAdd [b, b': Book, n: Name, a: Addr]{
  add [b, b', n, a]
  #Name.(b'.addr) > 1
}
run showAdd for 3 but 2 Book
```

Operações de remoção e busca

- Remoção: remove todas as associações entre um nome n e qualquer endereço
- Busca: qualquer conjunto de endereços para o qual o nome n está mapeado sob addr em b. Definida como uma função, não como um predicado (o corpo é uma expressão, não uma restrição)

```
pred del [b, b': Book, n: Name] {
    b'.addr = b.addr - n->Addr
}

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

Definition

Restrição que deveria ser válida, verdadeira, em todos os possíveis casos

Contra-exemplo

- Cenário em que uma asserção é violada
- Pode indicar uma falha no modelo
  - Semelhante a *erro* em código

Vamos utilizar asserções para definir o comportamento da combinação em sequência das operações add e del

```
assert delUndoesAdd {
  all b, b', b'': Book, n: Name, a: Addr |
   add [b, b', n, a] and del [b', b'', n]
      implies b.addr = b''.addr
}
check delUndoesAdd for 3
```

- Há um contraexemplo. Por quê?

Revendo a asserção delUndoesAdd, modificamos a asserção para os casos em que não há entradas para o nome n

```
assert delUndoesAdd {
    all b, b', b'': Book, n: Name, a: Addr |
        no n.(b.addr) and add [b, b', n, a]
                      and del [b', b'', n]
        implies b.addr = b''.addr
}
check delUndoesAdd for 10 but 3 Book
```

- Idempotência da adição

```
assert addIdempotent {
    all b, b', b'': Book, n: Name, a: Addr |
        add [b, b', n, a] and
        add [b', b'', n, a]
        implies b'.addr = b''.addr
}
check addIdempotent for 3
```

- Adição é uma operação local: adicionar uma entrada para um nome n não afeta o resultado de uma pesquisa para um nome n'

```
assert addLocal {
  all b, b': Book, n, n': Name, a: Addr |
    add [b, b', n, a] and n != n'
      implies lookup [b, n'] = lookup [b', n']
}
check addLocal for 3 but 2 Book
```

Aplicação real

- Pode-se criar um *alias* para um endereço, e então usar tal *alias* como alvo para um outro *alias*
- Um *alias* pode nomear múltiplos alvos. Podemos nos referir a um grupo de endereços com um único nome

Hierarquia de classificação

Modelo

```
module tour/addressBook2a

abstract sig Target { }
sig Addr extends Target { }
abstract sig Name extends Target { }

sig Alias, Group extends Name { }
sig Book {addr: Name -> Target}
```

Hierarquia de classificação

- Explorando o espaço de estado com simulação de predicados

```
pred show [b:Book]{ some b.addr }
run show for 3 but 1 Book
```

- Problema: um *alias* mapeado para si mesmo

Definition

Restrição que assumimos que é sempre verdadeira

Qualquer que seja o catálogo, não há nome que pertença ao conjunto de alvos alcançáveis a partir do próprio nome

```
fact{
  all b:Book | no n:Name | n in n.^(b.addr)
  }
```

Hierarquia de classificação

Fato

- Fatos que se aplicam a todos os membros de uma assinatura são escritos como *fatos de assinatura*

```
sig Book {addr: Name -> Target}
  {no n: Name | n in n.^addr}
```

- Resultado da simulação: grupo com dois endereços

Hierarquia de classificação

- Mapeando um *alias*

```
pred show [b:Book] {some Alias.(b.addr)}
run show for 3 but 1 Book
```

- Problema: um *alias* mapeado em dois endereços (deveria ser um grupo)
- Adicionamos novo fato

```
sig Book {addr: Name -> Target}
  {no n: Name | n in n.^addr
   all a: Alias | lone a.addr
  }
```

Hierarquia de classificação

- Ainda temos um problema: um *alias* pode ser mapeado em um grupo vazio
- A busca por um nome pode não resultar em endereços, mesmo com o nome no catálogo
  - Tornamos explícito o conjunto de nomes presentes no catálogo (campo *names*)
  - Modificamos a declaração de mapeamento do endereço para mapear apenas nomes deste conjunto

```
sig Book {
    names: set Name,
    addr: names -> some Target
} {
    no n: Name | n in n.^addr
    all a: Alias | lone a.addr
}
```

Hierarquia de classificação

- Nova asserção: toda pesquisa de um nome produz resultado

```
assert lookupYields {
    all b: Book, n: b.names | some lookup [b,n]
}
```

- Problema: busca vazia

Hierarquia de classificação

Atualização das operações de adição, remoção e busca

```
pred add [b, b': Book, n: Name, t: Target]
                    { b'.addr = b.addr + n->t }
pred del [b, b': Book, n: Name, t: Target]
                    { b'.addr = b.addr- n->t }
fun lookup [b: Book, n: Name] : set Addr
                    { n.^(b.addr) & Addr }
```

Traços de execução

Problema A função de busca pode resultar em um conjunto vazio de endereços

Solução

- Observar o efeito de múltiplos passos a partir do estado inicial
- Adicionar um ordenamento
  - Primeiro catálogo satisfaz algumas condições iniciais
  - Catálogos adjacentes são relacionados por uma operação

Traços de execução

- O predicado init define a condição inicial
- O fato traces especifica as restrições que fazem do ordenamento um traço

```
module tour/addressBook3a
open util/ordering [Book] as BookOrder
...
pred init [b: Book]  { no b.addr }

fact traces {
    init [first]
    all b: Book-last |
      let b' = b.next |
        some n: Name, t: Target |
          add [b, b', n, t] or del [b, b', n, t]
}
```

Traços de execução

- Obtendo um traço por meio de uma instância que satisfaz um predicado vazio

```
pred show ()
run show for 4
```

Traços de execução

- Problema da busca vazia

```
assert lookupYields { all b: Book, n: b.names | some lookup [b,n] }
check lookupYields for 3 but 4 Book
```

- O contra-exemplo mostra como uma sequência de operações leva a um estado indesejado

Traços de execução

- Problema: o predicado add permite um *alias*, que se refere a nada, ser adicionado a um grupo
- Solução: pré-condição: alvo deve ser um endereço ou ser resolvido para, pelo menos, um endereço na busca

```
pred add [b, b': Book, n: Name, t: Target] {
    t in Addr or some lookup [b,t]
    b'.addr = b.addr + n->t
}
```

- Novo problema: removemos o último membro de um grupo

Traços de execução

- Solução: proibir a remoção do último membro de um grupo por meio de uma pré-condição

```
pred del [b, b': Book, n: Name, t: Target] {
    no b.addr.n or some n.(b.addr) - t
    b'.addr = b.addr - n->t
}
```

O nome n não é mapeado ou mapeado em algum alvo que não seja t

Análise: **check** lookupYields **for** 6

Referências

- Livro: D. Jackson. Software Abstractions - Logic, Language, and Analysis. MIT Press, 2006.
- Página de Alloy: http://alloy.mit.edu/community/