Alloy Analyzer 4

Część 1: Wprowadzenie i logika

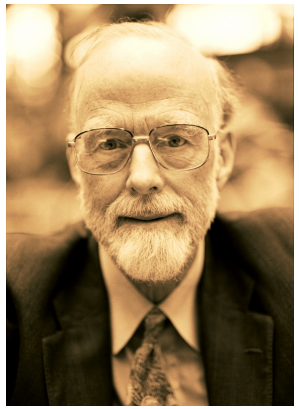
prezentacja oparta na materiałach autorstwa
Grega Dennisa i Roba Seatera
Software Design Group, MIT

dlaczego projektowanie deklaratywne?

„Dochodzę do wniosku, że istnieją dwa sposoby konstruowania projektu programistycznego.

Pierwsza polega na zapewnieniu, aby był on tak prosty, aby w oczywisty sposób nie było w nim błędów, a druga – na takim jego skomplikowaniu, aby nie było w nim oczywistych błędów."

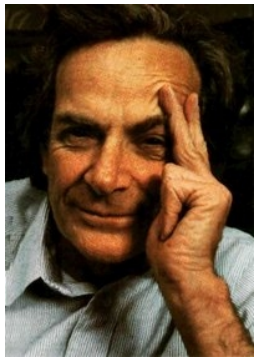
– Tony Hoare [wykład z okazji odbioru *Nagrody Turinga*, 1980]



dlaczego zautomatyzowana analiza?

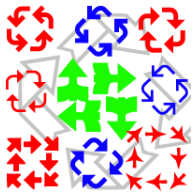
Podstawowa zasada – nie oszukuj sam siebie
i pamiętaj, że nikogo równie łatwo nie oszukasz.

– Richard P. Feynman



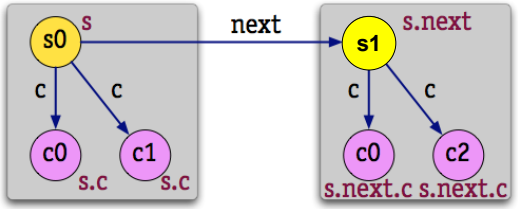
najważniejsze hasła …

- wszystko jest relacją
- niespecjalizowana logika
- kontrprzykłady i „zakres"
- analiza za pomocą SAT



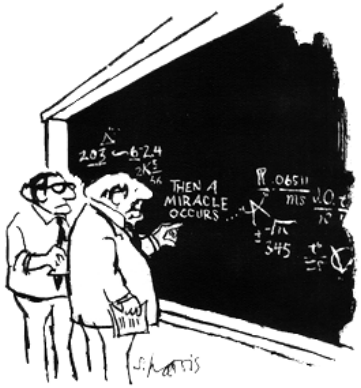
wszystko jest relacją

- Alloy używa relacji do modelowania wszystkiego
  - typów danych – nawet zbiorów, skalarów, krotek
  - struktur w czasie i przestrzeni :)
- podstawowy operator to „**kropka**”
  - złączenie relacji
  - odwoływanie się do „pól”
  - zastosowanie funkcji do argumentu



niespecjalizowana logika

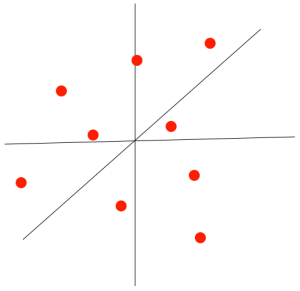
- Brak *dedykowanych* konstrukcji do opisu własności modeli – wykorzystujemy logikę predykatów



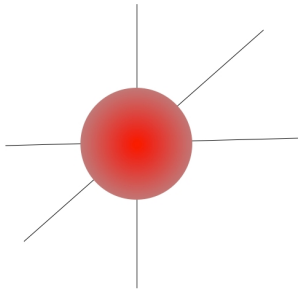
"I think you should be more explicit here in step two."

kontrprzykłady i zakres

- obserwacja wynikająca z analizy projektów:
  - większość założeń zawiera błędy
  - większość błędów ma „małe kontrprzykłady”



testowanie kilku „losowo wybranych” przypadków



analiza wszystkich przypadków w ograniczonym zakresie

analiza za pomocą SAT

- SAT
  - Problem *spełnialności* formuły φ rachunku zdań

$$\exists \mathbf{v} \,.\, \mathbf{v} \models \varphi \quad ?$$

  - SAT jest **NP-zupełny**

    żaden problem NP-zupełny nie posiada znanego algorytmu o wielomianowej złożoności (P != NP ?)

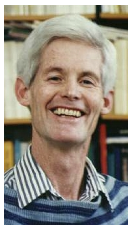
    nawet najlepsze algorytmy SAT mają wykładniczą złożoność w pesymistycznym przypadku

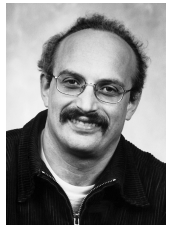
    na szczęście „pesymistyczne przypadki” zdarzają się rzadko

    algorytmy SAT radzą sobie z problemami wielkości milionów klauzul i setek tysięcy zmiennych zdaniowych w kilka sekund...

analiza za pomocą SAT

- SAT
  - Alloy redukuje zadanie weryfikacji modelu do weryfikacji SAT
  - może korzystać z różnych weryfikatorów SAT: BerkMin, MiniSat, SAT4J, ZChaff, ...


Stephen Cook


Eugene Goldberg


Henry Kautz


Sharad Malik


Yakov Novikov

modelujemy „sufity i podłogi”

```
sig Platform {}
```
będziemy mówili o „poziomach”

```
sig Man {ceiling, floor: Platform}
```
każdy człowiek ma sufit i podłogę

```
pred Above [m, n: Man] {m.floor = n.ceiling}
```
m „mieszka nad” n jeśli podłoga m jest sufitem n

```
fact {all m: Man | some n: Man | Above[n,m] }
```
„Każdy sufit jest czyjąś podłogą”

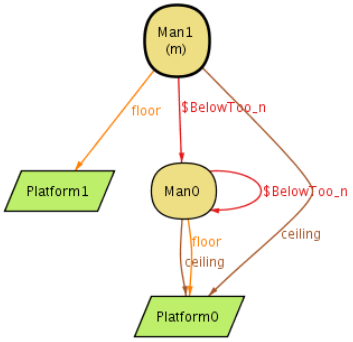
weryfikujemy „sufity i podłogi”

```
assert BelowToo {
  all m: Man | some n: Man | Above[m,n]
}
```
„Każda podłoga jest czyimś sufitem”?

```
check BelowToo for 2
```
sprawdzamy, czy „każda podłoga jest czyimś sufitem” - szukamy kontrprzykładu z 2 lub mniej ludźmi i poziomami

- kliknięcie „Execute” wykonuje powyższe polecenie
  - znajdujemy kontrprzykład...

kontrprzykład dla „BelowToo”



Alloy = logika + język + analiza

- logika
  - logika pierwszego rzędu + rachunek relacji
- język
  - składnia pozwalająca na strukturalizację specyfikacji w logice
- analiza
  - ograniczone i wyczerpujące poszukiwanie kontrprzykładu dla zadanej własności, wykorzystujące weryfikację SAT

logika: relacje na atomach

- atomy to byty pierwotne w Alloy-u
  - niepodzielne, niezmienne, nieinterpretowane
- relacje łączą atomy
  - zbiory krotek – ciągów atomów
- wszystkie wartości w logice Alloy-a to relacje!
  - relacje, zbiory, skalary są wszystkie tym samym

logika: wszystko jest relacją

- zbiory to relacje unarne (1 argumentowe)

```
Name = {(N0),     Addr = {(A0),     Book = {(B0),
        (N1),             (A1),             (B1)}
        (N2)}             (A2)}
```

- skalary to zbiory jednoelementowe

```
myName   = {(N1)}
yourName = {(N2)}
myBook   = {(B0)}
```

- relacje binarne

```
names = {(B0, N0),
         (B0, N1),
         (B1, N2)}
```

- relacje trójargumentowe

```
addrs = {(B0, N0, A0),
         (B0, N1, A1),
         (B1, N1, A2),
         (B1, N2, A2)}
```

logika: relacje

```
addrs = {(B0, N0, A0), (B0, N1, A1),
         (B1, N1, A2), (B1, N2, A2)}
```

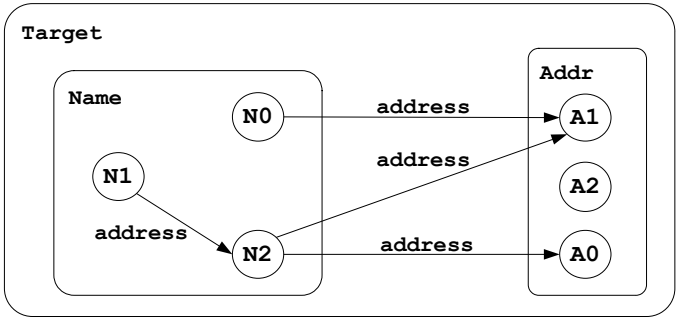
B0	**N0**	**A0**	size = 4
B0	**N1**	**A1**	
B1	**N1**	**A2**	
B1	**N2**	**A2**	

arity = 3

- wiersze są nieuporządkowane
- kolumny są uporządkowane
- wszystkie relacją są relacjami pierwszego rzędu
  - nie mogą zawierać innych relacji; nie ma rodzin zbiorów (zbiorów zbiorów), itd.

logika: przykład - książka adresowa

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
Target = {(N0), (N1), (N2), (A0), (A1), (A2)}
address = {(N0, A1), (N1, N2), (N2, A1), (N2, A0)}
```



logika: stałe

none	*zbiór pusty*
univ	*„uniwersum”*
iden	*relacja identycznościowa*

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

none = {}
univ = {(N0), (N1), (N2), (A0), (A1)}
iden = {(N0, N0), (N1, N1), (N2, N2),
        (A0, A0), (A1, A1)}
```

logika: operatory mnogościowe

+	*suma*
&	*przecięcie*
-	*różnica*
in	*podzbiór*
=	*równość*

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

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

```
greg = {(N0)}
rob = {(N1)}

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

```
cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}

cacheAddr + diskAddr = {(N0, A0), (N1, A1), (N1, A2)}
cacheAddr & diskAddr = {(N0, A0)}
cacheAddr = diskAddr = false
```

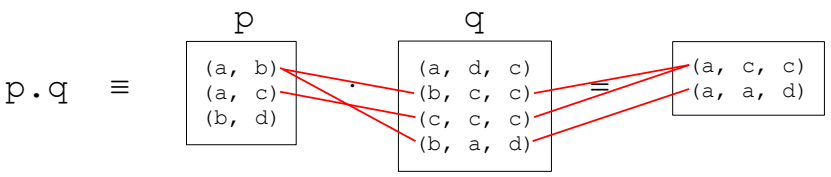
logika: operator produktu

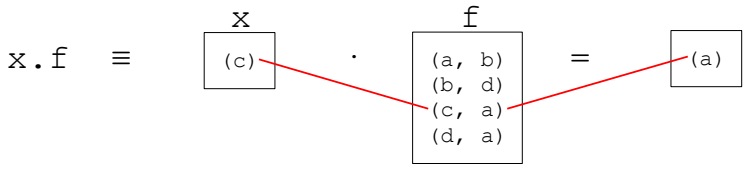
->	*produkt kartezjański*

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

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

```
b  = {(B0)}
b' = {(B1)}
address  = {(N0, A0), (N1, A1)}
address' = {(N2, A2)}

b->b' = {(B0, B1)}

b->address + b'->address' =
  {(B0, N0, A0), (B0, N1, A1), (B1, N2, A2)}
```

logika: złączenie relacji





logika: operatory złączenia

`.`	*dot join*
`[]`	*box join*

```
e1[e2] = e2.e1
a.b.c[d] = d.(a.b.c)
```

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

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

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

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

host[myAddr] = {(H0)}
address.host = {(B0, N0, H0), (B0, N1, H0), (B0, N2, H1)}
```

logika: operatory unarne

`~`	*transpozycja*
`^`	*domknięcie przechodnie*
`*`	*domknięcie zwrotne i przechodnie*

stosowane tylko do relacji binarnych

```
^r = r + r.r + r.r.r + …
*r = iden + ^r
```

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

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
         (N1, N2), (N1, N3),
         (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
         (N1, N1), (N1, N2), (N1, N3),
         (N2, N2), (N2, N3), (N3, N3)}
```

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

first.^next = rest
first.*next = Node
```

logika: restrykcje i przesłonięcia

`<:`	*restrykcja dziedziny*
`:>`	*restrykcja obrazu*
`++`	*przesłonięcie*

```
p ++ q =
p - (domain[q] <: p) + q
```

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

address :> Addr  = {(N2, A0)}
Alias <: address = address :> Name  = {(N0, N1), (N1, N2)}
address :> Alias = {(N0, N1)}

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

```
m' = m ++ (k -> v)
```

unacześnienie odwzorowania m przez parę klucz-wartość (k, v)

logika: operatory boolowskie

`!`	**not**	*negacja*
`&&`	**and**	*koniunkcja*
`||`	**or**	*dyzjunkcja*
`=>`	**implies**	*implikacja*
	else	*alternatywa*
`<=>`	**iff**	*równoważność*

cztery równoważne stwierdzenia:

```
F => G else H

F implies G else H

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

(F and G) or ((not F) and H)
```

logika: kwantyfikatory

```
all x: e | F
all x: e1, y: e2 | F
all x, y: e | F
all disj x, y: e | F
```

all	*zachodzi dla **każdego***
some	*zachodzi dla **co najmniej jednego***
no	*nie zachodzi dla **żadnego***
lone	*zachodzi dla **co najwyżej jednego***
one	*zachodzi dla **dokładnie jednego***

```
some n: Name, a: Address | a in n.address
```
pewnej nazwie przypisany jest pewien adres — książka adresowa jest niepusta

```
no n: Name | n in n.^address
```
do żadnej nazwy nie możemy dojść od niej samej — książka jest acykliczna

```
all n: Name | lone a: Address | a in n.address
```
każdej nazwie przypisany jest co najwyżej jeden adres — „funkcyjność”

```
all n: Name | no disj a, a': Address | (a + a') in n.address
```
inny sposób powiedzenia powyższego

logika: deklaracje zbiorów

```
x: m e
```

```
Q x: m e
```

```
x: e <=> x: one e
```

set	*dowolna liczba elementów*
one	*dokładnie jeden element*
lone	*co najwyżej jeden element*
some	*co najmniej jeden element*

```
RecentlyUsed: set Name
```
RecentlyUsed jest podzbiorem zbioru Name

```
senderAddress: Addr
```
senderAddress jest jednoelementowym podzbiorem Addr

```
senderName: lone Name
```
senderName jest pustym albo jednoelementowym podzbiorem Name

```
receiverAddresses: some Addr
```
receiverAddresses jest niepustym podzbiorem Addr

logika: deklaracje relacji

```
r: A m -> n B
Q r: A m -> n B
```

```
r: A -> B <=>
r: A set -> set B
```

```
(r: A m -> n B) <=>
   ((all a: A | n a.r) and (all b: B | m r.b))
```

```
workAddress: Name -> lone Addr
```
każdy alias odwołuje się do co najwyżej jednego adresu służbowego

```
homeAddress: Name -> one Addr
```
każdy alias odwołuje się do dokładnie jednego adresu domowego

```
members: Name lone -> some Addr
```
adres należy do co najwyżej jednej grupy,
a grupa zawiera co najmniej jeden adres

```
r: A -> (B m -> n C) <=>
all a: A | a.r: B m -> n C
```

```
r: (A m -> n B) -> C <=>
all c: C | r.c: A m -> n B
```

logika: wyrażenia kwantyfikowane

some e	*e zawiera **co najmniej jedną** krotkę*
no e	*e nie zawiera **żadnej** krotki*
lone e	*e zawiera **co najwyżej jedną** krotkę*
one e	*e zawiera **dokładnie jedną** krotkę*

```
some Name
```
zbiór nazw jest niepusty

```
some address
```
książka adresowa jest niepusta – zawiera jakąś krotkę

```
no (address.Addr - Name)
```
w książce adresowej adresom przypisane są jedynie nazwy

```
all n: Name | lone n.address
```
w książce adresowej każdej nazwie przypisany jest co najwyżej jeden adres

logika: wyszczególnienia

```
{x1: e1, x2: e2, ..., xn: en | F}
```

```
{n: Name | no n.^address & Addr}
```
zbiór nazw, którym nie odpowiada żaden adres

```
{n: Name, a: Address | n -> a in ^address}
```
relacja binarna łącząca nazwy i „osiągalne” adresy

logika: wyrażenia warunkowe i let

```
f implies e1 else e2
let x = e | formuła
let x = e | wyrażenie
```

cztery równoważne stwierdzenia:

```
all n: Name |
  (some n.workAddress
   implies n.address = n.workAddress
   else n.address = n.homeAddress)

all n: Name |
  let w = n.workAddress, a = n.address |
    (some w implies a = w else a = n.homeAddress)

all n: Name |
  let w = n.workAddress |
    n.address = (some w implies w else n.homeAddress)

all n: Name |
  n.address = (let w = n.workAddress |
    (some w implies w else n.homeAddress))
```

logika: liczby i kardynalności

`#r`	*liczba krotek w r*
`0,1,...`	*literały całkowite*
`n.plus[…]`	*plus*
`n.minus[…]`	*minus*

`=`	*równe*
`<`	*mniejsze*
`>`	*większe*
`=<`	*mniejsze lub równe*
`>=`	*większe lub równe*

```
sum x: e | ie
```
suma wartości wyrażenia całkowitego ie dla x pochodzących z e

```
all d: Dir | #d.files =< 3
```
wszystkie katalogi mają po nie więcej niż 3 pliki

```
#File = sum d: Dir | #d.files
```
liczba wszystkich plików jest równa sumie liczb plików we wszystkich katalogach

dwie logiki w jednej

- „wszyscy kochają zwycięzców”
- logika predykatów
  - $\forall$`w | Winner(w)` $\Rightarrow$ $\forall$`p | Person(p)` $\Rightarrow$ `loves(p, w)`
- rachunek relacyjny
  - `Person` $\times$ `Winner` $\subseteq$ `loves`
- Logika Alloy-a – obie możliwości
  - **all** `w: Winner, p: Person | p -> w` **in** `loves`
  - `Person -> Winner` **in** `loves`
  - **all** `p: Person | Winner` **in** `p.loves`