Logika dla informatyków

Andrzej M. Borzyszkowski

Instytut Informatyki
Uniwersytet Gdański

sem. zimowy 2025/26

`inf.ug.edu.pl/~amb`

w oparciu o tutorial "Practical Alloy" `https://practicalalloy.github.io/`

Modelowanie softwar'u – Alloy

Modelowanie softwar'u

- specyfikacja: zbiór formuł, które określają zakładane okoliczności użycia softwaru'e
- model: zbiór z działaniami spełniający specyfikację chcemy/możemy sprawdzać własności modeli
- czy istnieje model specyfikacji? — niesprzeczność
- czy dla każdego modelu specyfikacji zachodzi pewna własność? rozpatrujemy wyłącznie skończone modele — tylko takie mogą być reprezentowane w komputerze możemy zbadać tylko skończenie wiele modeli — np. o rozmiarze $\leq$ pewna liczba

Alloy

- system wspomagania modelowania software'u opracowany w MIT
  - deklaratywny — użytkownik opisuje własności
  - może generować przykłady/ kontrprzykłady specyfikacji
- podstawy matematyczne: naiwna teoria mnogości, relacje (niemal) jak w SQL: atom to zbiór 1-elementowy, zbiór to tablica 1-kolumnowa, relacja to dowolna tablica
- założenie pragmatyczne: jeśli coś jest źle, mały kontrprzykład to pokaże
- a więc należy wyczerpująco zbadać zbiór małych modeli

Przykład wiodący: system plików

- pliki umieszczone są w katalogach, katalogi zawierają pliki i katalogi
  - tzn. relacje „ojciec" i „zawartość"
- katalog główny jest jedyny i nie należy nigdzie, pozostałe katalogi umieszczone są w innych katalogach
- w systemie występują wyłącznie pliki i katalogi (oczywiście jest to uproszczeniem)
- plik nigdy nie jest katalogiem i na odwrót
- chcemy sprawdzić czy istnieją kontrprzykłady na stwierdzenia
  - tylko jeden katalog nie ma ojca
  - system katalogów nie ma cyklu
  - każdy katalog należy co najwyżej do jednego podkatalogu

Specyfikacja systemu plików

```
abstract sig Object {} //obiekty
sig Dir extends Object { entries : set Entry } //katalog ma zawartosc
sig File extends Object {} //o pliku nic wiecej
one sig Root extends Dir {} //jest jeden Root
sig Entry { object : one Object, name   : one Name } // obiekty beda
    zwiazane z nazwami
sig Name {}
```


Podstawy matematyczne

```
abstract sig E { }
sig S extends E { F: one T }
fact { all e:S | e.F in X }
```

E jest klasą	E jest zbiorem
S jest pod podklasą E	S jest podzbiorem zbioru E
F jest polem w S typu T	F jest relacją binarną, odwzorowuje elementy S w elementy T, tutaj funkcja
e jest instancją klasy S	e jest elementem zbioru S
. (kropka) wybiera pola	. jest złączeniem relacji
e.F jest elementem typu T należącym do X	e.F jest złączeniem relacji unarnej e z binarną F, wynik jest relacją unarną w T zawiera się w X

Język specyfikacji

- extends — nowa klasa, rozłączna z dotychczasowymi
- in — niekoniecznie rozłączna

```
fact restrict_object { Object in Dir + File }
fact no_shared_entries {
  all disj x, y : Dir | no (x.entries & y.entries)
  all e : Entry | lone entries.e }
fact one_directory_per_entry {
  all e : Entry | one entries.e }
fact unique_names {
  all d : Dir, disj x, y : d.entries | x.name != y.name
  all d : Dir, n : Name | lone (d.entries & name.n) }
fact no_dangling_objects {
  Entry.object = Object - Root }
```

- one x (dokładnie jeden), lone x (zero lub jeden), no x (nie ma)

Asercje

- fact jest przesłanką, narzuceniem warunku na modele, tylko takie rozpatrujemy
- assert jest warunkiem, chcielibyśmy by był implikacją semantyczną wszystkich faktów
- Alloy szuka kontrprzykładów dla niewielkich modeli

```
fact  no_self_containment {
  all d : Dir | d not in d.entries.object }
fun descendants [o : Object] : set Object {
  o.^(entries.object) }
assert no_indirect_containment {
   all d : Dir | d not in descendants [d] }
check no_indirect_containment
```

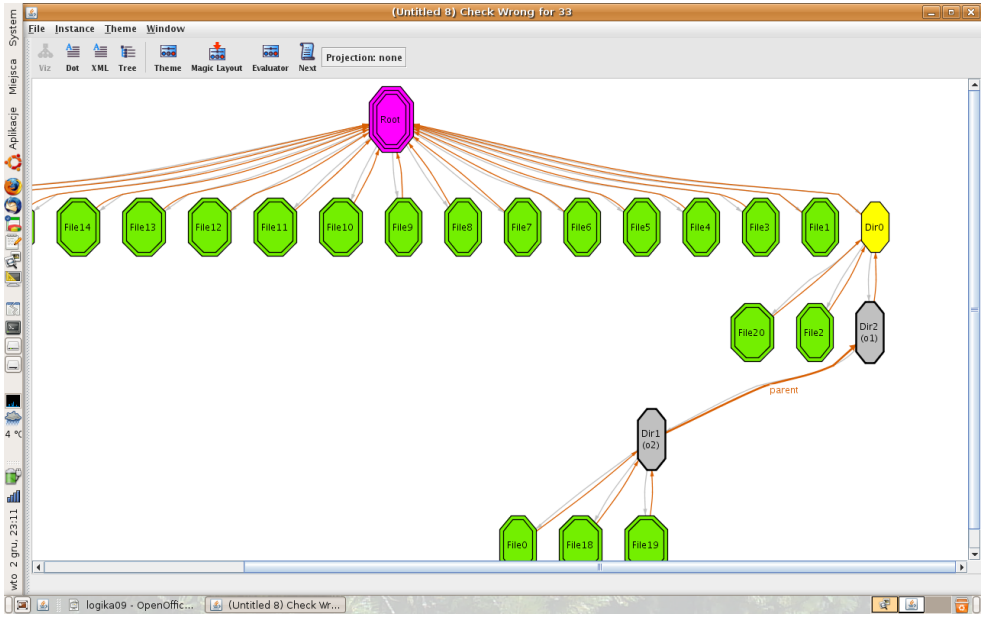
- zabezpieczyliśmy się przed krótką pętlą ale nie przed dłuższym cyklem

Asercje – dalszy przykład

```
assert AllBro {
all o1, o2: (Object − Root)
| (o1.parent = o2.parent) }
// kazde dwa obiekty maja wspólnego
     ojca
check AllBro for 4
```

- system wyprodukuje kilka kontrprzykładów już dla 3 elementów, dla większej liczby też

Fantazyjny kontrprzykład



System plików – dalsza rozbudowa

- chcemy teraz opisać możliwość dokonywania zmian
- w przykładzie pliki będą umieszczane w repozytorium, będą mogły być udostępniane innym użytkownikom, którzy będą mogli je pobierać, będą mogły być usuwane, tzn. umieszczane w koszu i z niego usuwane lub odtwarzane

```
sig Token {}
sig File { var shared : set Token }
var sig uploaded in File {}
var sig trashed in uploaded {}
fact init { //init is just a label assigned to the fact,
  no uploaded
  no shared }
```

- słowo kluczowe var oznacza, że zawartość tych klas/zbiorów będzie się zmieniać

Specyfikacja repozytorium plików

- możliwe akcje
  - wstaw plik do repozytorium
  - usuń plik (czyli wstaw do kosza)
  - odtwórz plik z kosza
  - opróżnij kosz (cały)
  - nadaj plikowi token (jednorazowy, przeznaczony dla jednego użytkownika w celu jednorazowego pobrania pliku)
  - pobierz plik w/g danego tokena
- akcje będą opisane poprzez predykaty mówiące w jakich warunkach mogą być wywołane i jakie skutki spowodują
- będą to jedyne możliwe akcje

Specyfikacja repozytorium plików, c.d.

- ```
  fact transitions { always (
    (some f: File | upload [f] or delete [f] or restore [f]) or
    (some f: File, t :Token | share [f,t]) or
    (some t: Token | download [t]) or
    empty
  ) }
  ```

```
pred upload [f: File] {
  f not in uploaded          // guard
  uploaded' = uploaded + f   // effect on uploaded
  trashed' = trashed         // no effect on trashed
  shared' = shared           // no effect on shared
}
pred delete [f: File] {
  f in uploaded − trashed       // guard
  trashed' = trashed + f        // effect on trashed
  shared' = shared − f−>Token   // effect on shared
  uploaded' = uploaded          // no effect on uploaded
}
```


Specyfikacja repozytorium plików, c.d.

- ```
  pred restore [f: File] {
    f in trashed                  // guard
    trashed' = trashed − f        // effect on trashed
    uploaded' = uploaded          // no effect on uploaded
    shared' = shared              // no effect on shared
  }
  pred share [f: File, t: Token] {
    f in uploaded − trashed           // guard
    historically t not in File.shared  // guard
    shared' = shared + f−>t           // effect on shared
    uploaded' = uploaded              // no effect on uploaded
    trashed' = trashed                // no effect on trashed
  }
  pred download [t: Token] {
    some shared.t               // guard
    shared' = shared − File−>t   // effect on shared
    uploaded' = uploaded        // no effect on uploaded
    trashed' = trashed          // no effect on trashed
  }
  ```


Specyfikacja repozytorium plików, c.d.

- ```
  pred empty {
    no trashed'                  // effect on trashed
    uploaded' = uploaded − trashed   // effect on uploaded
    shared' = shared                 // no effect on shared
  }
  ```

- to jeszcze nie jest dobra specyfikacja, bo nie zakłada możliwości, że nic się nie dzieje
  trzeba taką możliwość przewidzieć i dodać do specyfikacji tranzycji

- ```
  pred stutter {
    uploaded' = uploaded    // no effect on uploaded
    trashed' = trashed      // no effect on trashed
    shared' = shared        // no effect on trashed
  }
  run example {}
  ```

Logika temporalna czasu liniowego, również przeszłości

- słowa kluczowe logiki temporalnej (liniowej):
  - always, eventually, after, releases
  - historically, once, before, triggers

```
assert shared_are_accessible {
  always shared.Token in uploaded − trashed }
//check shared_are_accessible
assert restore_undoes_delete {
  all f : File | always (
    delete [f] and after restore [f] implies
    uploaded'' = uploaded and trashed'' = trashed and shared'' = shared
  ) }
//check restore_undoes_delete
```

- pierwsza asercja jest prawdziwa, druga nie, Alloy znajduje kontrprzykład

Dalsze przykłady spójników temporalnych

- ```
  assert one_download_per_token {
    all t : Token | always ( download [t] implies after always not download [t]
        )
    all t : Token | always lone downloaded [t]
  }
  fun downloaded [t : Token] : set File {
    { f : File | once (t in f.shared and download[t]) }
  }
  //check one_download_per_token
  ```
- asercja jest prawdziwa, są dwa sposoby jej zapisu

Dalsze przykłady spójników temporalnych c.d.

- ```
  assert empty_after_restore {
    all f: File | always (
      delete [f] implies
      after ((restore [f] or upload [f]) releases not delete [f])
    )
  }
  //check empty_after_restore
  ```
- wszystkie przykłady dotyczyły własności bezpieczeństwa – gwarantujemy, że zawsze zachodzi pożądana własność, czyli nigdy nie zdarzy się niepożądana

Przykład działania

- ```
  run behavioral_modeling_instance {
    some f0 : File, disj t0, t1 : Token {
      File  = f0
      Token = t0 + t1
      upload[f0];share[f0, t1];share[f0, t0];download[t1];delete[f0];restore[f0]
      after after after after after always (delete[f0] implies after restore[f0])
      after after after after after always (restore[f0] implies after delete[f0])
  }}
  ```
- zostanie wygenerowany system tranzycji z kilkoma akcjami, w końcówce jest pętla, która wykonuje naprzemian usunięcie pliku i jego odtworzenie
  ```

Sprawdzanie żywotności *liveness*

- ```
  assert non_restored_files_will_disappear {
    all f : File | always ( delete [f] and after always not restore [f] implies
      eventually f not in uploaded
    ) }
  //check non_restored_files_will_disappear
  ```
- ta asercja nie jest prawdziwa, plik po usunięciu może pozostać w koszu na zawsze
- specyfikacja może żądać aktywnie, by pewne akcje wystąpiły
- ```
  fact fairness_on_empty { always eventually empty }
  ```
- akcja empty jest zawsze możliwa, *fairness* mówi, że akcja, która jest zawsze możliwa, kiedyś powinna być wykonana
  czyli przebieg, który systematycznie jej nie wykonuje jest niedopuszczalny

Rodzaje *fairness*

- ```
  pred fairness_on_empty { always eventually empty
  }
  pred strong_fairness_on_empty {
    (always eventually some trashed) implies always eventually empty
  }
  pred weak_fairness_on_empty {
    always ( always ( some trashed ) implies always eventually empty )
  }
  ```
- pierwszy warunek może być zbyt ograniczający – żąda bezwzględnie opróżniania kosza, nawet gdy nic w nim nie ma
  gdyby akcja empty wymagała czegoś w koszu, byłoby to po prostu niewykonalne
- drugi warunek też może nie być praktyczny, żąda, że jeśli coś się w koszu pojawi to kosz powinien być opróżniony
  ale użytkownik mógłby bez przerwy coś do kosza wrzucać i wyjmować bez opróżniania

Rodzaje *fairness* c.d.

- sprawdzenie, czy warunki fairness gwarantują żądany wynik
- ```
  check { non_restored_files_will_disappear }
  check {
    fairness_on_empty implies non_restored_files_will_disappear
  }
  check {
    strong_fairness_on_empty implies non_restored_files_will_disappear
  }
  check {
    weak_fairness_on_empty implies non_restored_files_will_disappear
  }
  ```
- uwaga: warunki fairness ograniczają system tranzycji, wpływa to na wydajność systemu, lepiej nie narzucać tych warunków tylko sprawdzać implikacje j.w.