Rozszerzenia JML-a do weryfikacji programów wielowątkowych

Aleksander Lewandowski

20 maja 2009

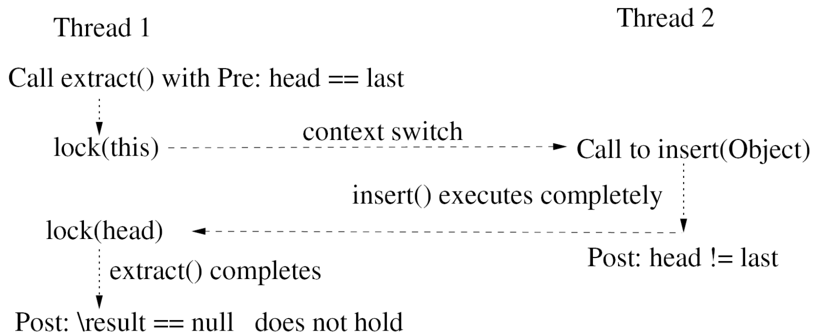
Plan prezentacji

1. Postawienie problemu
   - Możliwości JML-a
   - Problematyczne przeploty
   - Wyrażanie zachowań bezpiecznych w programach wielowątkowych
2. Pojęcia atomowości i niezależności
   - Idea atomowości
   - Teoria redukcji Liptona
   - Atomowość i niezależność w teorii Liptona
3. Wprowadzanie współbieżności do JML-a
   - Notacje już istniejące w JML-u
   - Blokady
   - Ograniczanie dostępu do sterty
   - Modyfikatory metod, blokowanie i transakcje
4. Podsumowanie

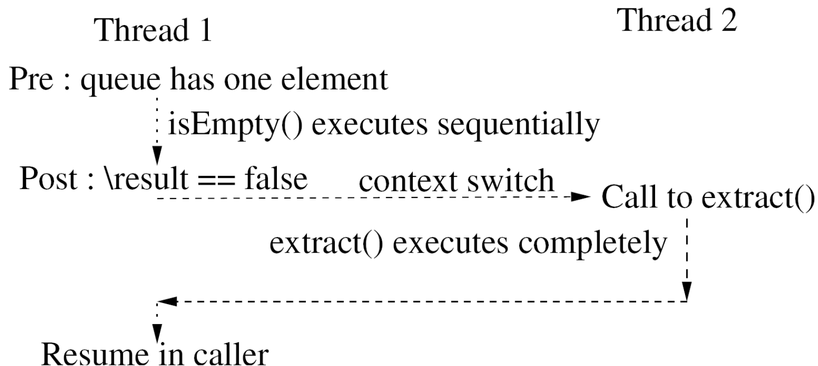
Specyfikacja JML-owa metody *extract()*

```
public class LinkedQueue {
  protected /*@ spec_public non_null @*/ LinkedNode head;
  protected /*@ spec_public non_null @*/ LinkedNode last;
  //@ public invariant head.value == null;
  /*@ public normal_behavior
    @   requires head == last;
    @   assignable \nothing;
    @   ensures \result == null;
    @ also public normal_behavior
    @   requires head != last;
    @   assignable head, head.next.value;
    @   ensures head == \old(head.next) && \result == \old(head.next.value);
    @*/
  public synchronized Object extract() {
    synchronized (head) {
      Object x = null;
      LinkedNode first = head.next;
      if (first != null) {
        x = first.value;
        first.value = null;
        head = first;
      }
      return x;
    }
  }
```

Internal Interference



External Interference



Blokady

- Blokady które metoda zdobędzie i zwolni w czasie wykonania
- Blokady chroniące części stanów obiektów
- Obiekty używane jako blokady i momenty w których te obiekty są zablokowane
- Zbiory blokad trzymanych przez wątek
- Fakt ochrony obiektu poprzez blokadę trzymaną przez obecny wątek

Specyfikacja ograniczeń danych

- Jakie aliasy obiektu istnieją w środowisku
- Kto jest właścicielem obiektu
- Lokalność obiektu względem metody/wątku
- Efekty wykonania metody na istniejących lokacjach

Idea atomowości

- Sekwencję instrukcji wykonywaną przez dany wątek nazwiemy atomową jeżeli dla każdego wykonania programu zawierającego tę sekwencję (być może przeplataną wykonaniami instrukcji innych wątków) istnieje równoważne wykonanie w którym instrukcje rozważanej sekwencji wykonywane są bezpośrednio po sobie.

Teoria redukcji Liptona

- Fragment kodu, będący sekwencją prymitywnych instrukcji wykonywanych przez jeden wątek, nazywamy tranzycją.
- Intuicyjnie: prawym (lewym) spychaczem (*right/left mover*) nazywamy taką tranzycję α, dla której jeśli następnikiem (poprzednikiem) α jest tranzycja β należąca do innego wątku, to α i β mogą zostać zamienione i nie będzie miało to wpływu na stan wynikowy.
- Tranzycją spychającą (*commuting transition*) nazywamy trakzycję która jest lewym lub prawym spychaczem.

Teoria redukcji Liptona c.d.

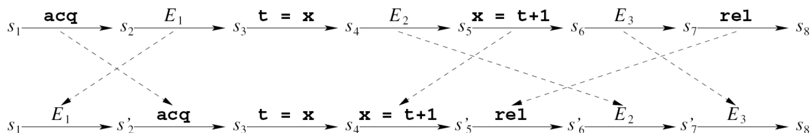
Program *P*, zawierający sekwencję **tranzycji** *S*, jest równoważny programowi *P/S*, w który *S* jest zamodelowane jako pojedyncza niepodzielna tranzycja

wtedy i tylko wtedy gdy

zbiór stanów końcowych P jest równy zbiorowi stanów końcowych *P/S*

Przykład

Rozważmy metodę *m*, która zajmuje blokadę, czyta zmienną chronioną przez tę blokadę, zmienia jej wartość po czym zwalnia blokadę. Załóżmy, że kolejne kroku tej metody są przeplecione z tranzycjami E_1, E_2, E_3 wykonywanymi przez inne wątki. Ponieważ akcje metody *m* są spychaczami (zajęcie i zwolnienie blokady odpowiednio lewymi i prawymi, przypisanie do chronionej zmiennej obustronnym) więc istnieje równoważne wykonanie metody nie przeplecione z operacjami innych wątków.



Atomowość i niezależność w teorii Liptona

- Atomowym fragmentem kodu nazwiemy fragment pasujący do wzorca $R^* N^? L^*$, gdzie R^* oznacza zero lub więcej tranzycji będących prawymi spychaczami, L^* zero lub więcej tranzycji będących lewymi spychaczami, a $N^?$ zero lub jedną tranzycję nie będącą żadnym spychaczem.
- Niezależnym fragmentem kodu nazwiemy fragment pasujący do wzorca I^*, gdzie I^* oznacza zero lub więcej tranzycji będących obustronnymi spychaczami.

Notacje już istniejące w JML-u

- Aby uzyskać dostęp do zmiennej <ident> wszystkie blokady wymienione w <store-ref-list> muszą być trzymane przez wątek:

  <monitors-for-clause> ::= `monitors_for` <ident> <- <store-ref-list> ;

- `\lockset()`: obiekt typu `JMLObjectSet` reprezentujący zbiór wszystkich blokad trzymanych przez wątek.

`locks` $l_1, \ldots, l_n$

<locks-clause> ::= `locks` <store-ref-list> ;

- Występuje w ciele specyfikacji po nagłówku (po `requires`).
- Stwierdza, że metoda blokuje (i zwalnia) tylko te blokady które są wymienione.
- Zauważmy, że zachodzi:
  `ensures \old(\lockset().has(l1) && ... && \lockset().has(ln) ==>\independent;`
- Dzięki powyższej implikacji mamy dolne ograniczenie na zbiór blokad potrzebnych do tego, żeby metoda była niezależna.

lock_protected o

<lock-protected-expression> ::=
`lock_protected`(<store-ref>);

- Dostęp do obiektu wskazywanego przez o jest chroniony niepustym zbiorem blokad i wątek trzyma je wszystkie.
- Chroniony jest obiekt, nie wskaźnik.

Modyfikator dostępu `rep`

- Do użycia w deklaracjach pól w klasach.
- Poza obiektem klasy nie może być referencji do tego miejsca na stercie.
- Czasem użycie takiego modyfikatora może umożliwić modularną weryfikację atomowości...

Przykład

```
public class BetterLinkedQueue {
  protected /*@ spec_public non_null rep @*/ LinkedNode head;
  protected /*@ spec_public non_null rep @*/ LinkedNode last;
  //@ public invariant head.value == null;

    /*@ public normal_behavior
     @  requires head == last;
     @  locks this, head;
     @  assignable \nothing;
     @  ensures \result == null;
     @  also public normal_behavior
     @  requires head != last;
     @  locks this, head;
     @  assignable head, head.next.value;
     @  ensures head == \old(head.next) && \result == \old(head.next.value);
    @*/
  public /*@ atomic @*/ synchronized /*@ readonly @*/ Object extract() {
    synchronized (head) {
      /*@ readonly @*/ Object x = null;
      /*@ rep @*/ LinkedNode first = head.next;
      if (first != null) {
        x = first.value;
        first.value = null;
        head = first;
      }
      return x;
    }
  }
```

Modyfikator dostępu `readonly`

- Do użycia we wszystkich deklaracjach referencji do obiektów.
- Obiekt wskazywan przez oznaczoną referencję nie może byc **przy jej użyciu** modyfikowany. Przez aliasy może.

`thread_local o`

<thread-local-expression> ::= `thread_local`(<store-ref>);

- Tylko dany wątek może osiągnąć `o` poprzez łańcuch referencji.
- Użyteczne w modularnym weryfikowaniu atomowości, ponieważ dostęp do zminnych lokalnych dla wątku jest niezależny.
- Zapewnia, że *race condition* nie wystąpi.
- Pomocniczy operator: `thread_safe(o)` $\equiv$ `thread_local(o)` $\|$ `lock_protected(o)`

Przykład

```
/*@ normal_behavior
  @   requires c != null && \thread_local(c);
  @   assignable elementCount, elementData;
  @   ensures elementCount == c.size() && \fresh(elementData);
  @also
  @ exceptional_behavior
  @   requires c == null;
  @   assignable \nothing;
  @   signals (Exception e) e instanceof NullPointerException;
  @*/
public /*@ atomic @*/ Vector(Collection c) {
  elementCount = c.size();
  elementData = new Object[(int)Math.min((elementCount*110L)/100,Integer.MAX_VALUE)];
  c.toArray(elementData);
}
```

Modyfikator `atomic`

- Jeśli metoda zostanie wywołana w stanie spełniającym warunki wstępne to musi spełniać wymagania atomowości w sensie Liptona.
- Eliminuje przeploty 'internal interference'.
- Może 'dziedziczyć sekcję krytyczną'. Przykład...

Dziedziczenie blokady - przykład

```
public class ArrayBlockingQueue<E> {
  private /*@ spec_public non_null rep @*/ final E[] items;
  //@ monitors_for items <- lock;
  private /*@ spec_public rep @*/ final ReentrantLock lock;
  /*@ normal_behavior
    @   requires lock.isLocked() && 0 < i && i < items.length;
    @   ensures \result == \old((i + 1) % items.length) && \independent;
    @*/
  final /*@ atomic @*/ int inc(int i) {
    return (++i == items.length)? 0 : i;
  }
```

Prefykat `\independent`

- Metoda jest niezależna, wtedy kiedy wszystkie tranzycje są niezależne.
- Używany tylko w klauzuli `ensures`.
- Przykłady: operacje na zmiennych lokalnych względem wątku, dostęp do zmiennych zablokowanych na rzecz wątku
- Przykładowy kod...(poprzedni slajd)

Czekanie na spełnienie warunku

- Predykat `when`:
  <when-clause> ::= `when` <predicate> ;
- Znacznik `commit`

Czekanie na spełnienie warunku - przykład

```
/*@ public normal_behavior
  @   locks this.lock;
  @   when count != 0;
  @   assignable items[takeIndex], takeIndex, count;
  @   ensures \result == \old(items[takeIndex]) && takeIndex == \old(takeIndex + 1)
  @        && count == \old(count - 1);
  @*/
public /*@ atomic @*/ E take() throws InterruptedException {
  final ReentrantLock lock = this.lock;
  lock.lockInterruptibly();
  try {
    try {
      while (count == 0)
        notEmpty.await();
    } catch (InterruptedException ie) {
      notEmpty.signal(); // propagate to non-interrupted thread
      throw ie;
    }
    /*@ commit: @*/ E x = extract();
    return x;
  } finally {
    lock.unlock();
  }
}
```

Inne

Należy pamiętać, że projekt pochodzi z czasów Javy 1.5.

- Obsługa klas dziedziczących po interfejsie `java.utli.concurrent.locks.Lock` - jest.
- Obsługa klasy `java.util.concurrent.ReentrantLock` - jest.

Statystyki wystąpień rozszerzeń

Class Name	Number of methods	Frequency of annotations				
		atomic	\independent	locks	\thread_safe	when
BoundedBuffer	3	3	0	2	0	2
DiningPhilosphers	7	7	4	2	0	1
LinkedQueue	7	7	0	7	0	1
RWVSN	8	8	2	4	0	2
java.util.Vector	45	45	4	34	9	0
ArrayBlockingQueue*	19	19	7	15	3	2
CopyOnWriteArrayList*	27	27	6	13	12	0
CopyOnWriteArraySet*	13	13	2	6	5	0
DelayQueue*	17	17	3	14	4	2
LinkedBlockingQueue*	17	17	4	12	1	2
PriorityBlockingQueue*	21	21	4	10	1	1
ConcurrentLinkedQueue*	11	11	2	0	2	4
Total:	**195**	**195**	**38**	**119**	**37**	**17**

Klasy oznaczone * pochodzą z pakietu `java.util.concurrent` z Javy 1.5

Statystyki powodzenia weryfikacji rozszerzeń

Class Name	Number of Methods	Checkable Atomicity	Checkable Functionality	Coverage Ratio
`BoundedBuffer`	3	1	3	.67
`DiningPhilosphers`	7	6	7	.93
`LinkedQueue`	7	1	7	.57
`RWVSN`	8	4	8	.75
`CopyOnWriteArrayList`*	10	5	10	.75
`LinkedBlockingQueue`*	7	3	7	.71
Total:	**42**	**20**	**42**	**.74**

Klasy oznaczone * pochodzą z pakietu `java.util.concurrent` z Javy 1.5