Techniki zabezpieczania kodu – sprawdzanie modeli

Temat VI

Podstawowa idea

- Skończony model oprogramowania +
- Formuła logiki temporalnej (LTL, CTL, μ-rachunek...)
  Czy w systemie nie ma dead-locku?
  Czy wysłany pakiet w końcu zostanie odebrany?
- $\Rightarrow$
- Ślad potencjalnego błędu lub OK

Własności temporalne

- Własności bezpieczeństwa:
  - niezmienniki,
  - deadlocki
  - osiągalność itp.

  sprawdzenie na skończonych śladach
- Własności żywotności
  - sprawiedliwość
  - responsywność itp.

  sprawdzenie na nieskończonych śladach

Zastosowanie

- Błąd w Pentium Intela był “katastrofą”, która wprowadziła sprawdzanie modeli do przemysłu tworzenia sprzętu
- Intel, IBM, Motorola, itp. zatrudniają obecnie setki ekspertów od sprawdzania modeli
- Co wprowadzi sprawdzanie modeli do przemysłu wytwórstwa oprogramowania?

Możliwe podejścia do stanu

- Stan jawny
- Stan symboliczny

Stan jawny

- Idea:
  - stany tworzone w locie,
  - analiza ruchów do przodu,
  - odwiedzone stany zapisane w tablicy haszującej.

Stan jawny

- Własności:
  - pamięciożerność,
  - sprawdza się przy znajdowaniu błędów współbieżności,
  - krótkie ścieżki wykonania są lepsze, ale długie też można obsługiwać
  - możliwa obsługa tworzenia obiektów/wątków
- Zwykle używane do analizy oprogramowania

Stan symboliczny

- Idea:
  - w każdej chwili operacje wynonywane na zbiorach stanów
  - zwykle analiza wstecz,
  - relacja przejścia zakodowana zwykle jako
    * (pewien wariant) BDD (binary decision diagram) lub
    * problem spełnialności

Stan symboliczny

- Własności:
  - jest w stanie obsługiwać bardzo duże przestrzenie stanów,
  - niezbyt dobra dla systemów asynchronicznych
  - nie radzi sobie dobrze z długimi ścieżkami wykonania
  - najlepiej działa ze statyczną relacją tranzycji, więc słabo daje sobie radę z dynamicznym tworzeniem obiektów/wątków.
- Zwykle używane do analizy sprzętu

Problemy związane ze sprawdzaniem modeli

- konstruowanie modeli
- opisywanie własności
- eksplozja liczby stanów
- interpretacja wyników

Problemy z konstruowaniem modeli

Przepaść semantyczna:

- Języki programowania:
  metody, dziedziczenie, dynamiczna alokacja, wyjątki, itp.
- Język modeli:
  automaty skończone

Problemy z konstruowaniem modeli

- Analiza stosowalna do małych jednostek kodu (eksplozja stanów)
- Niejasne granice między jednostkami:
  - referencje wypływają poza jednostkę,
  - zewnętrzne komponenty mogą zmieniać stan obiektów z jednoski,
  - powszechna obecność call-backów (np. w kodzie GUI),
  - pracochłonne definiowanie punktów interakcji

Problemy z opisywaniem własności

- *Między otwarciem a zamknięciem okna przycisk X może być przyciśnięty najwyżej dwa razy*
- odpowiednia formuła LTL:

```
[]((open /\ <>close) ->
     ((!pushX /\ !close) U
    (close \/ ((pushX /\ !close) U
     (close \/ ((!pushX /\ !close) U
              (close \/ ((pushX /\ !close) U
               (close \/ (!pushX U close)))))))))
```

Problemy z opisywaniem własności

- Własności specyfikuje się w terminach modelu, a nie kodu źródłowego
- Obiekty mają różne identyfikatory w trakcie życia

Tłumaczenie półautomatyczne

- Tłumaczenie i abstrakcja oparte o tablice
  - system Feaver Gerarda Holzmanna
  - specyfikacja fragmentów kodu w C i sposobu ich translacji do języka abstrakcyjnego (Promela w SPINie)
  - reszta translacji automatyczna
  - znaleziono 75 błędów w systemie PathStar firmy Lucent http://cm.bell-labs.com/cm/cs/who/gerard/
- Zalety:
  - można użyć ponownie po zmianie programu
  - działa dobrze dla programów o długim czasie tworzenia i lokalnych zmianach

Tłumaczenie w pełni automatyczne

- Zaleta – udział człowieka niepotrzebny
- Wada – ograniczone możliwości systemu docelowego
- Przykłady:
  - Java PathFinder
    http://ase.arc.nasa.gov/havelund/jpf.html
    Java $\Rightarrow$ Promela (Spin)
  - JCAT
    http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml
    Java $\Rightarrow$ Promela (lub dSpin)
  - Bandera
    http://www.cis.ksu.edu/santos/bandera/
    bajtkod Javy $\Rightarrow$ Promela, SMV or dSpin

Problemy z eksplozją stanów

- Dołożenie równoległej jednostki (zwykle obiektu) powoduje pomnożenie przestrzeni stanów przez rozmiar proporcjonalny do tej jednostki.
- Negacja formuły powoduje wykładniczy przeskok
- Pomaga prawo Moore'a (wzrost mocy komputerów)
- ...ale problem ze skalowalnością zostaje.

Przykładowa technika redukcji stanów

Redukcje częściowo-porządkowe w JPF

- Wykorzystanie stałych zbiorów (ang. persistent sets)
- Wynajdywanie tranzycji, które są globalnie niezależne
- Wymaga analizy statycznej przed sprawdzeniem modelu, aby określić niezależność (analiza aliasów)

Problemy z interpretacją wyników

- długie ślady błędów,
- przepaść semantyczna, sprytne kodowania, wielokrotne optymalizacje i transformacje
- abstrakcja powoduje powstawanie nadmiarowych śladów

Abstrakcja

- Sprawdzanie modeli zwykle nie działa na "prawdziwych programach"
- Sprawdzanie modeli zwykle działa na systemach skończeniestanowych
- Abstrakcja rozwiązuje dwa problemy:
  - pozwala sprawdzać coś, na czym sprawdzaczka nie może bezpośrednio działać
  - obcina przestrzeń stanów do rozsądnych rozmiarów

Abstrakcja

- Trzy rodzaje abstrakcji:
  - przybliżenia z góry – do systemu dodane są nowe zachowania
  - przybliżenia z dołu – po abstrakcji mamy mniej zachowań
  - abstrakcje precyzyjne – w zabstrahowanym i oryginalnym programie są te same zachowania

Przybliżanie z dołu

- Usuwamy części programu, które uważamy za "nie mające związku" ze sprawdzaną własnością
  - ograniczenie wartości do 0..10 zamiast całego `int`
  - rozmiar kolejki 3 zamiast nieograniczonego itp.
- Początkowo najpopularniejsza forma sprawdzania modeli
- Zwykle wykonywana ręcznie bez gwarancji, że usuwane są tylko dobre zachowania
- Możliwa do uzyskania abstrakcja precyzyjna
- Przekroje programu jako przykład zautomatyzowanego przybliżania od dołu (ang. program slicing)

Przybliżanie z góry

- Odwzorowywanie zbiorów stanów konkretnego programu w pojedyncze stany programu abstrakcyjnego
- Mniejsza liczba stanu, ale więcej możliwych tranzycji, a więc i zachowań
- W rzadkich wypadkach może prowadzić do precyzyjnej abstrakcji
- Abstrakcje oparte na typach (`int` zastąpiony przez $\{NEG, ZERO, POS\}$)

Przybliżanie z góry c.d.

- Abstrakcja predykatów (zastępujemy predykaty zmiennymi boolowskimi a instrukcje modyfikujące wartość predykatu modyfikującymi wartość zmiennych)
- Automatyczna (konserwatywna) abstrakcja
- Duży problem z eliminacją nadmiarowych błędów
- Odnoszenie błędów w programach abstrakcyjnych do oryginalnych programów

Przykład abstrakcji typów

Abstrakcja typów danych w Banderze

- Na przykład: zamiast typu `int` rozważamy typ $\{NEG, ZERO, POS\}$
- Zdefiniowane abstrakcje w Bandera:
  - Range(i,j)
  - Modulo(k), Set(v,?)
  - Point – wszystko w jeden punkt
  - abstrakcje tablic – indeksy i elementy
  - abstrakcje klas – abstrakcje dla poszczególnych pól

Pokrycie testami w sprawdzaniu modeli

- Rzeczywistość:
  Przy sprawdzaniu modeli CZĘSTO brakuje pamięci
- Jeśli nie znaleziono błędu, to nie ma pewności, że go nie ma.
- Potrzebna jest miara pokrycia
- JPF został rozszerzony o obliczanie pokrycia rozgałęzień
- To pokrycie pozwala na lepsze heurystyczne poczukiwanie

Techniki w Java Path Finder/Banderze

- Abstrakcja predykatów i abstrakcja oparta o typy
- Redukcje częściowo-porządkowe i symetryczne
- Przekroje programów
- Przeszukiwanie heurystyczne
- Wykonywanie symboliczne
- Wyjaśnienia błędów

Case study – DEOS

Honeywell Digital Engine Operating System (DEOS)

- System czasu rzeczywistego dla zintegrowanych modularnych systemów lotniczych
- DEOS Scheduler: nietrywialny program w Javie: 1443 LOC, 20 klas, 6 wątków
- Własność:

  Procesy aplikacji mają gwarancję zaszeregowania przez opłaconą ilość czasu w trakcie trwania czasowej jednostki szeregowania.

Analiza dla K9 Mars Rover

- Rover to 8000 LOC przy 6 wątkach
  - dużo synchronizacji między wątkami,
  - złożone operacje na kolejkach
- Porównanie z tradycyjnym testowaniem
- Pierwotny kod w C++, potem przetłumaczony do Javy i C
- 4 grupy po 2 ludzi używało różnych technik wykrywania zadanych błędów

Metodologia dla sprawdzania modeli

- Zespół był proszony, aby nie uruchamiać kodu (tylko sprawdzać modele)
- Kod źródłowy opierał się mocno na zależnościach czasowych
- Zależności czasowe zamodelowane jako niedeterministyczny wybór
- Znaleziono wszystkie oprócz jednego znane błędy w programowaniu rozproszonym
- Lepiej niż pozostałe grupy
- Tylko ta grupa była w stanie wskazać, jak dojść do błędów
- Znaleziono także wszystkie błędy spoza programowania rozproszonego

Metodologia dla sprawdzania modeli

- Obserwacje:
  - Porzucono abstrakcję czasu w ciągu pierwszej godziny na korzyść rozwiązania bliższego czasowi rzeczywistemu, ale które mogło ukryć błędy
  - Zbyt trudne okazało się decydowanie, czy błędy są fałszywe, bez gruntownej znajomości kodu
  - Dużo czasu zajęło przygotowanie kodu, ale potem analiza szła szybko nawet po zmianach w kodzie

Główne narzędzia i ośrodki

1. Bandera (SAnToS Laboratory, Kansas State University)
2. Java PathFinder – JPF (NASA Ames)
3. SLAM Project (Microsoft Research)
4. BLAST Project (U. Berkeley)
5. FeaVer Project (Lucent/Bell Labs)
6. Alloy (MIT)
7. VeriSoft (dla C, Bell Laboratories, Lucent Technologies)