Chess

Joanna Iwaniuk

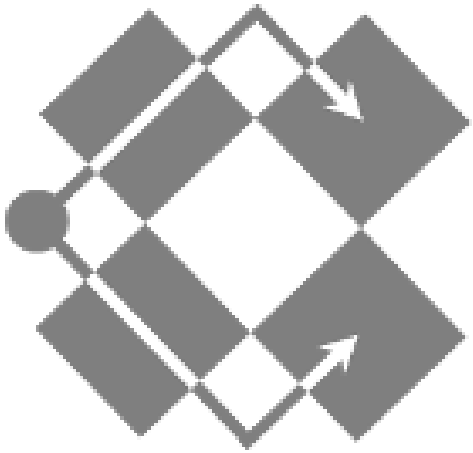
9 marca 2010

Plan prezentacji

1. Co to jest Chess?
2. Jak używać Chess?
3. Prezentacja działania
4. Jak to działa?
   - kontrola przeplotów
   - model checking
   - odtwarzanie wadliwego wykonania
5. Podsumowanie

Co to jest Chess?

- program Microsoftu służący do testowania programów współbieżnych
- cel: wykrywanie tzw. heisenbugs
- wersje dla różnych platform, przede wszystkim .NET
- w zasadzie darmowy ;)



Motywacja

- testowanie programów współbieżnych jest trudne
- tradycyjne podejście to stress testing
- nie mamy pewności, że zostały pokryte wszystkie przeploty
- błędy mogą pozostać niewykryte bardzo długo
- nawet jeśli wykryjemy błąd to jak go odtworzyć?

Postulaty

- Chess jako narzędzie do wykrywania błędów specyficznych dla programów współbieżnych
- testowanie deterministyczne
- pokrycie wszystkich (prawdopodobnych) przeplotów - model checking
- możliwość odtworzenia znalezionego błędu

Wyzwania

- Skąd wziąć model?
- Jak pozbyć się niedeterminizmu?
- Jak systematycznie przeszukiwać przeploty?
- Co zrobić z eksplozją stanów?
- Jak odtwarzać przeplot, który prowadzi do błędu?

Co wykryje Chess? (1)

- ogólnie: przypadki, kiedy test się nie powiódł
- ale: bynajmniej nie należy oczekiwać, że będzie to standardowy model checking
- zakładamy, że wykonanie pojedynczego wątku jest zawsze dokładnie takie samo

Co wykryje Chess? (2)

Dla różnych przeplotów, ale dla jednego scenariusza:

- podstawowe błędy takie jak segmentation fault (testowany program jest wykonywany)
- niespełnienie warunków testu
- zakleszczenia
- żywotność
- można zażądać wykrywania data races

Chess i C#

- potrzebna klasa ChessTest, która dostarcza scenariusz testowy
- ChessTest.Startup
- ChessTest.Run - powtarzane w pętli
- ChessTest.Shutdown

Prezentacja działania...

Jak działa Chess - najogólniejsze spojrzenie

```
TestStartup();
Chess.Quiesce();

while(true){
  RunTestScenario();
  Chess.Quiesce();
  if(Chess.Done()) break;
}

TestShutdown();
```

Przypomninenie

- Skąd wziąć model?
- Jak pozbyć się niedeterminizmu?
- Jak systematycznie przeszukiwać przeploty?
- Co zrobić z eksplozją stanów?
- Jak odtwarzać przeplot, który prowadzi do błędu?

Model *happens-before*

Graf *happens-before*

- Chess tworzy abstrakcję wszystkich wątków, timerów itp. - traktuje wszystkie takie jednostki jednakowo, jako zadania
- program jako automat niedeterministyczny z przejściami między stanami powodowanymi przez zadania
- węzły – aktualne instrukcje wykonywane przez dany wątek
- krawędzie – chronologiczny porządek między wykonanymi instrukcjami

Program – automat niedeterministyczny

- niedeterminizm związany z przeplotami
- inne rodzaje niedeterminizmu – zmienność danych wejściowych, random() itd.

Niedeterminizm vs Chess

- Chess przeszukuje kolejne ścieżki bez pamiętania całego grafu, poprzez powtarzanie wykonania testu
- potrafi zmusić program do przejścia dokładnie zadaną ścieżką – przeszukiwanie przestrzeni, odtwarzanie błędnego wykonania
- wniosek: potrzebne jest zapanowanie nad niedeterminizmem

1. Inne rodzaje niedeterminizmu

- rozwiązanie: obejście problemu :)
- dane wejściowe – powtarzanie tego samego scenariusza
- losowanie, odczytywanie informacji o czasie rzeczywistym, (...) – zapamiętywanie i deterministyczne odtwarzanie wyniku bez ponownego wywoływania funkcji

2. Niedeterminizm przeplotów

- ingerencja w system operacyjny?
- wirtualna maszyna?
- zmiana kodu źródłowego testowanego programu?
- trudne i kosztowne...
- rozwiązanie: własny scheduler Chess

Scheduler Chess

- wyręcza system operacyjny w kontroli przeplotów
- to trudne: trzeba umożliwić wszystkie oryginalnie możliwe przeploty, ale nie więcej
- kolejki zadań aktywnych, nieaktywnych, kontrola zmiennych synchronizacyjnych, pamiętanie które wątki zależą od których zmiennych...
- założenie: jednowątkowość

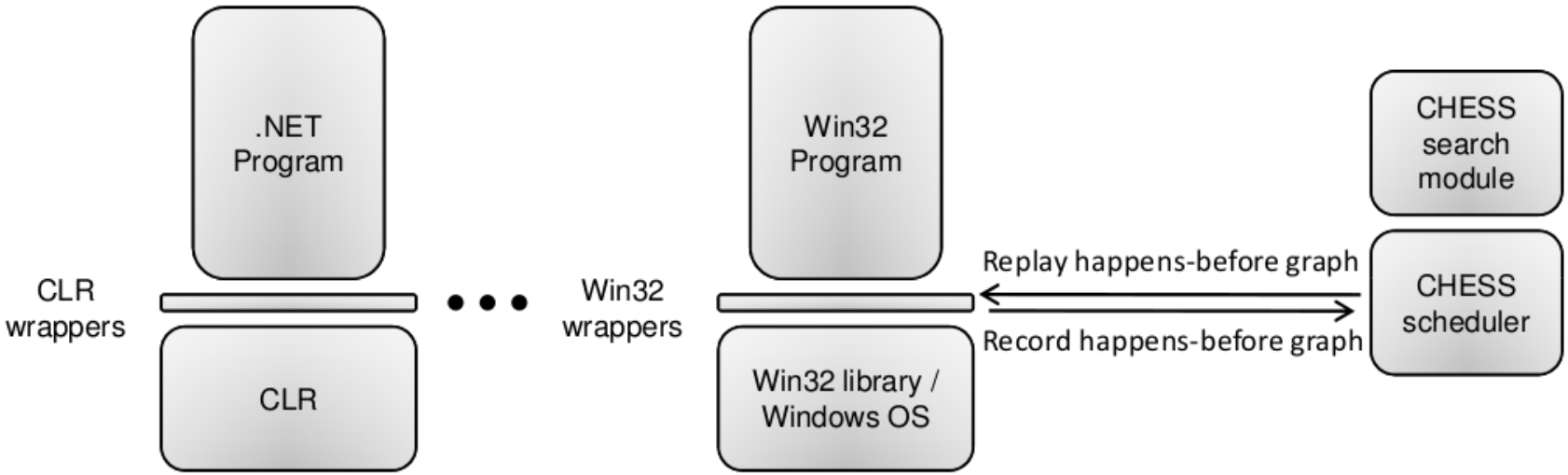
Jak to kontrolować?

- trzeba przechwycić informacje dla systemu operacyjnego
- Chess podmienia funkcje wołane przez program na swoje wersje - wrappery
- ogólnie: rola wrapperów – uchwycenie wystarczającego zakresu semantyki oryginalnego API, żeby umożliwić zbudowanie abstrakcji *happens-before*

Wrappery

- np. tworzenie nowego wątku – wrapper informuje Chess o powstaniu nowego wątku i wywołuje standardową funkcję
- wejście do sekcji krytycznej – wrapper używa wersji *try* oryginalnej funkcji, ewentualnie przenosi dany wątek do kolejki nieaktywnych
- potrzebna odrębna warstwa wrapperów dla każdej platformy (na szczęście dość cienka, np. dla .NET 64 wrappery, 1270 linii kodu)

Architektura Chess



Odtwarzanie przeplotu

- decyzje związane z przełączaniem kontekstu są zapamiętywane – graf *happens-before* ze wszystkimi zdarzeniami synchronizacyjnymi
- informacje można później wykorzystać do znalezienia kolejnego przeplotu lub odtworzenia poprzedniego
- problem – to nie są wszystkie informacje, są też inne źródła niedeterminizmu
- nie przejmujemy się tym specjalnie i jeśli nie można odtworzyć przeplotu to próbujemy powtórzyć test

Bezstanowy model checking

- model *happens-before* opisany wcześniej
- w pamięci nie jest przechowywany cały graf, a jedynie ostatnia ścieżka
- sprawiedliwy model checking w roli schedulera
- priorytety wątków

Formalnie

Oznaczenia predykatów i funkcji:

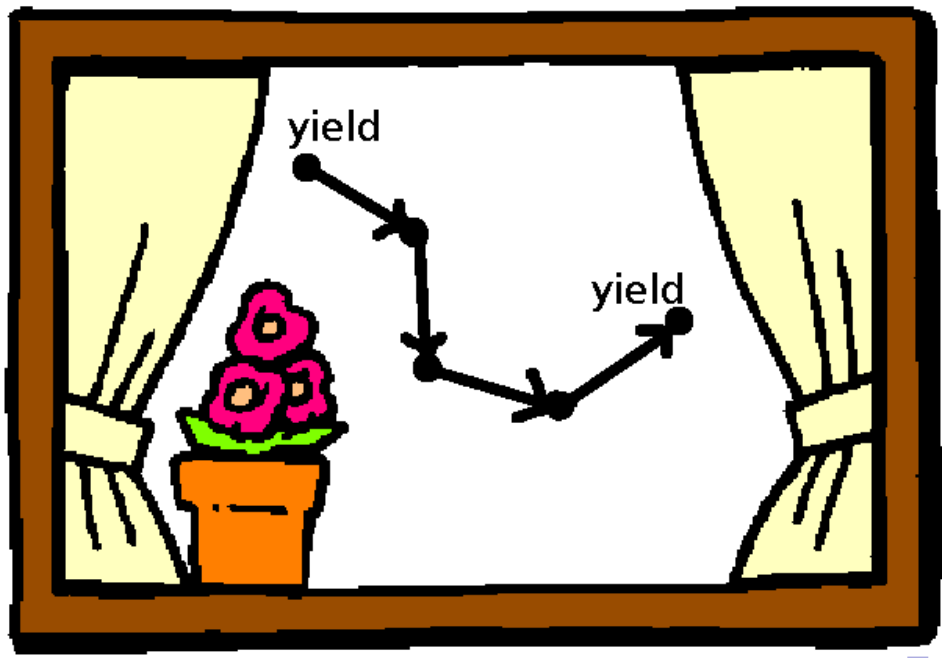
- *enabled(t)* – wątek t jest aktywny
- *yield(t)* – wątek t zaraz zrzeknie się procesora
- *sched(t)* – ostatnio wykonana instrukcja z wątku t
- ES(t) – abiór wątków aktywnych w danym stanie
- NextState(s, t) – stan do którego dojdziemy ze stanu wyznaczając t jako kolejny wątek do wykonania
- P – relacja priorytetu

Definicje własności

- sprawiedliwość: $\forall_{t \in Tid}\, \mathbf{GF}\, enabled(t) \Rightarrow \mathbf{GF}\, sched(t)$
- „własność dobrego samarytanina”
  $\forall_{t \in Tid}\, \mathbf{GF}\, sched(t) \Rightarrow \mathbf{GF}(sched(t) \wedge yield(t))$

Idea okienek

- okienko między dwoma stanami ze zrzeczeniem się procesora
- funkcje o interpretacji w przestrzeni okienka:
- S(t) – wątki przynajmniej raz wybrane (scheduled)
- E(t) – wątki przez cały czas aktywne (enabled)
- D(t) – wątki niaktywne (disabled)



Algorytm (pseudo-pseudokod)

curr := obecny stan := stan początkowy
loop forever
    T := curr.ES - wątki o niższym priorytecie
    if t = ∅ **then return**
    t := wybierz wątek z T (do wykonania)
    next := NextState(curr, t)
    next.P := curr.P - priorytety faworyzujące t
    foreach u ∈ wątki
        aktualizuj E, D, S dla t
    if curr.yield(t) **then**
        inicjalizuj E, D, S dla nowego okienka
        next.P := next.P ∪ te które nie zostały wybrane w
                poprzednim okienku mają wyższy priorytet niż t
    curr := next

Własności algorytmu

- każda nieskończona ścieżka wygenerowana przez algorytm ma własność **dobry samarytanin** $\Rightarrow$ **sprawiedliwość**.
- jeśli nie istnieje nieskończona ścieżka o powyższej własności to algorytm kończy się
- pozwala wykryć naruszenie własności dobrego samarytanina i zakleszczenie

Liczba wszystkich przeplotów

- dla n wątków i k kroków w wątku

$$n^k$$

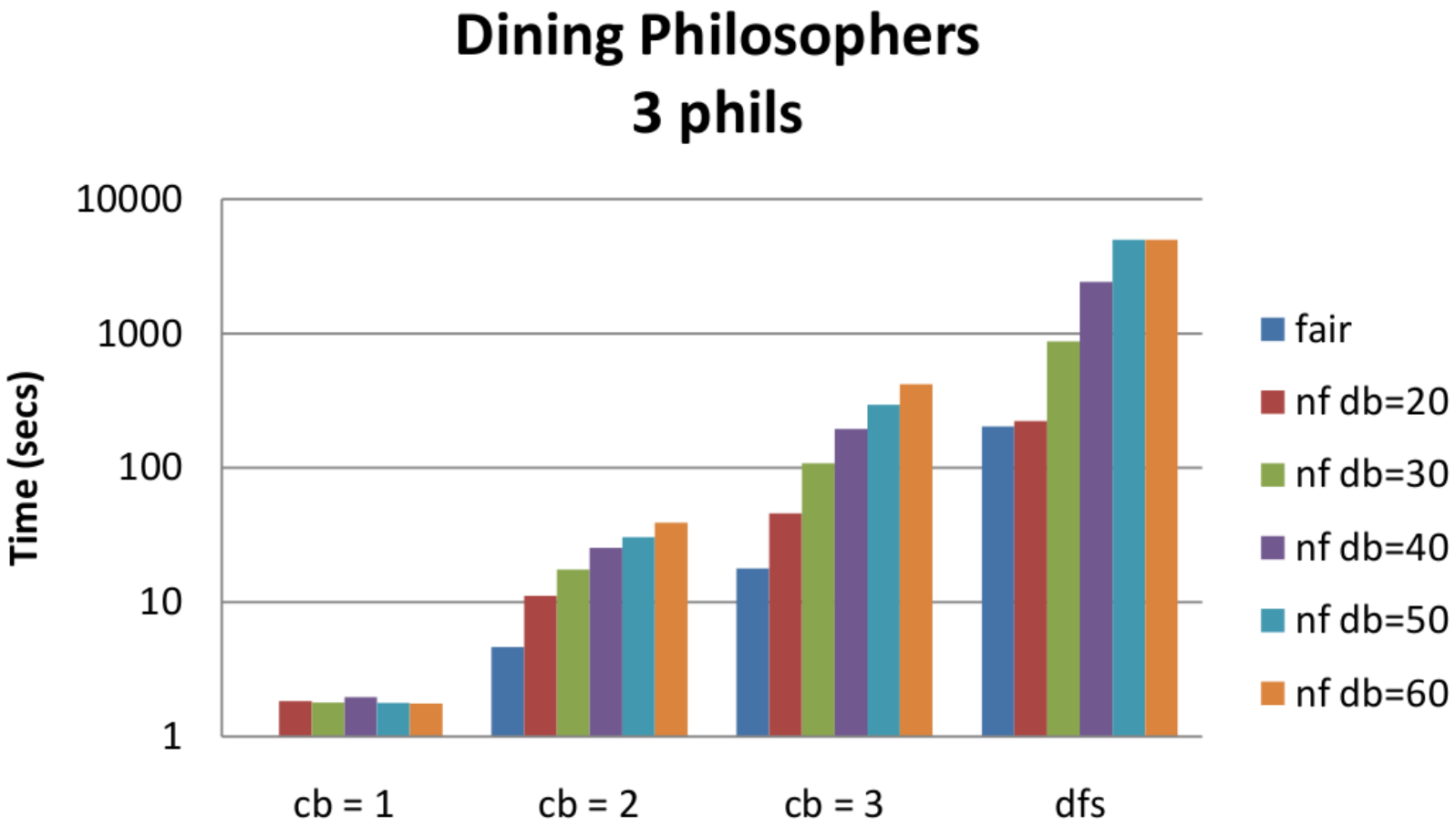
Preemption bounding

- ograniczenie analogiczne do ograniczenia głębokości przeszukiwania w zwykłym model checkingu
- ograniczenie na liczbę wywłaszczeń (liczba uśpień dowolna)
- rozwiązanie oparte na wierze, że błędy są powodowane przez nieliczne wywłaszczenia, ale w odpowiednich miejscach

Cechy preemption bounding

- jeśli liczba wywłaszczeń ograniczona do c, to rozmiar przestrzeni k^c (k – liczba kroków w wątku)
- dodatkowe heurystyki zmniejszające k
- stopniowe zwiększanie ograniczenia – znajdziemy najmniejsze c takie, dla którego jest źle, a zatem najprostszy przypadek, kiedy program nie działa

Wydajność model checkingu w różnych wersjach



Zakres stosowalności Chess

Programs	LOC	max Threads	max Synch.	max Preemp.
PLINQ	23750	8	23930	2
CDS	6243	3	143	2
STM	20176	2	75	4
TPL	24134	8	31200	2
ConcRT	16494	4	486	3
CCR	9305	3	226	2
Dryad	18093	25	4892	2
Singularity	174601	14	167924	1

Rysunek: Charakterystyka programów testowanych z powodzeniem za pomocą Chess

Statystyki skuteczności

Programs	Total	Failure / Bug		
		Unk/Unk	Kn/Unk	Kn/Kn
PLINQ	1		1	
CDS	1		1	
STM	2			2
TPL	9	9		
ConcRT	4	4		
CCR	2	1	1	
Dryad	7	7		
Singularity	1		1	
Total	27	21	4	2

Rysunek: Liczba błędów znalezionych przez Chess dla różnych programów

Zalety Chess

- w sumie znaleziono 25 nieznanych błędów (2007)
- niektóre z nich znalezione przez „niezależnych testerów”
- znaleziono błędy, które nie ukazały się przy stress testingu
- podobno Chess wykrył wszystkie błędy, które znaleziono przy stress testingu