NSWI162: Sémantika programů

7. Reálné systémy pro verifikaci pomocí kontraktů

Jan Kofroň

MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

Department of
Distributed and
Dependable
Systems

Verifikace programů

- Poznali jsme systém PiVC, který umožňuje verifikovat vlastnosti jednoduchého imperativního jazyka Pi
- Podobné nástroje existují i pro „skutečné“ programovací jazyky
  - Tyto jazyky jsou ale daleko složitější, a tak i jejich verifikace má své limity
  - Často verifikace nepostihuje nějakou část jazyka (objekty na haldě, real-time-ové vlastnosti, obecné cykly, …)

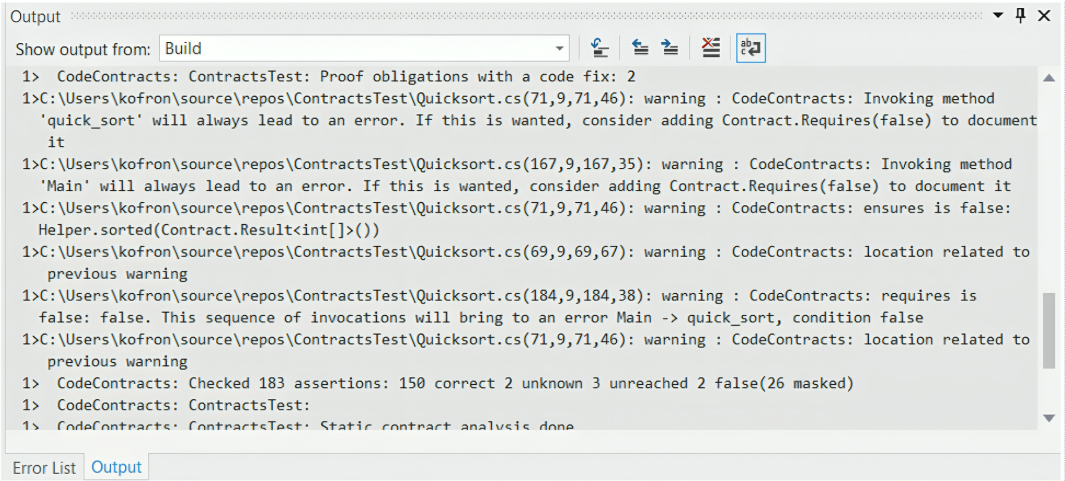
Code Contracts

- Systém anotací pro programovací jazyky nad platformou .NET
- Vyvinut Microsoftem, uvolněn jako open source
- Používá podobný styl anotací jako v PiVC
  - Preconditions, postconditions, invariants
- Podporuje i anotace pro případ, kdy nastane výjimka
- Anotace je možné ověřit staticky (při překladu) i kontrolovat za běhu
- Statické ověření během překladu není úplné
  - Když se verifikátoru nepodaří najít protipříklad ani dokázat platnost, nevíme nic, což se bohužel stává u složitějších případů často
- Dá se integrovat do VisualStudia (2013, omezeně i do 2017)

Code Contracts – Příklad

```
1   #define CONTRACTS_FULL
2
3   using System;
4   using System.Diagnostics.Contracts;
5
...
        3 references
74      public int[] qsort(int[] arr, int l, int u)
75      {
76          Contract.Requires(arr != null);
77          Contract.Requires(l >= 0);
78          Contract.Requires(u < arr.Length);
79          Contract.Requires(Helper.partitioned(arr, 0, l - 1, l, u));
80          Contract.Requires(Helper.partitioned(arr, l, u, u + 1, arr.Length - 1));
81
82          Contract.Ensures(Helper.sorted(Contract.Result<int[]>()));
83          Contract.Ensures(arr.Length == Contract.Result<int[]>().Length);
84          Contract.Ensures(Helper.eq(arr, Contract.Result<int[]>(), 0, l - 1));
85          Contract.Ensures(Helper.eq(arr, Contract.Result<int[]>(), u + 1, arr.Length - 1));
86          Contract.Ensures(Helper.partitioned(Contract.Result<int[]>(), 0, l - 1, l, u));
87          Contract.Ensures(Helper.partitioned(Contract.Result<int[]>(), l, u, u + 1, arr.Length - 1));
88
89          if (l >= u)
90          {
91              return arr;
92          }
```

Code contracts – Výstup



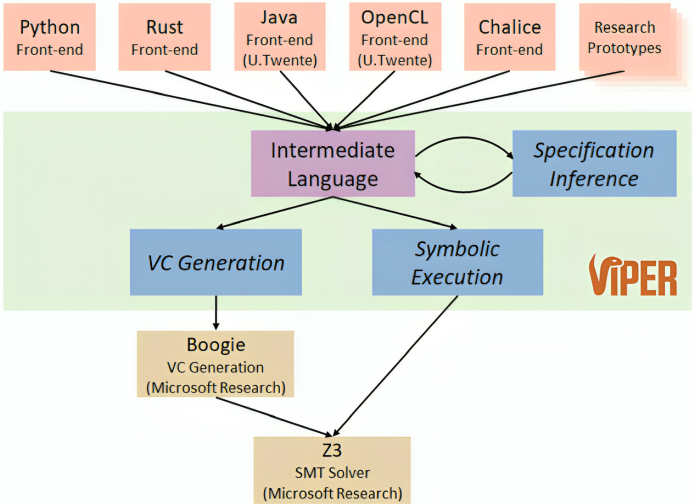
Code contracts – Další informace

- Code Contracts byly vyvinuty v rámci výzkumného projektu v Microsoftu
- Nejlepší zdroj informací o principech fungování jsou tedy vědecké články
- … anebo webové stránky

Nagini

- Nagini je front-end pro jazyk Python pro verifikaci kontraktů pomocí nástroje Viper
- Viper definuje svůj mezijazyk (zhruba na úrovni MS CIL), který pak verifikuje
- Existují front-endy i pro další jazyky
  - Ty se mi ale nepodařilo otestovat
- Stejně jako v předchozím případě programátor specifikuje kontrakty ve formě preconditions, postconditions a invariants
- Navíc obsahuje i podporu pro verifikaci vlastností vícevláknových programů
  - Včetně explicitních anotací korespondujících s vytvořením, join operací, a podobně

Nagini



Zdroj: `http://www.pm.inf.ethz.ch/research/viper.html`

NAGINI – PŘÍKLAD

```
from typing import List, cast
from nagini_contracts.contracts import *
from nagini_contracts.obligations import MustTerminate


def quickSort(arr: List[int]) -> List[int]:
    Requires(Acc(list_pred(arr), 2/3))
    Requires(MustTerminate(2 + len(arr)))
    Ensures(Acc(list_pred(arr), 2/3))
    Ensures(Implies(len(arr) > 1, list_pred(Result())))
    Ensures(Implies(len(arr) <= 1, Result() is arr))
    less = []  # type: List[int]
    pivotList = []  # type: List[int]
    more = []  # type: List[int]
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            Invariant(list_pred(less) and list_pred(pivotList) and list_pred(more))
            Invariant(len(Previous(i)) == len(less) + len(more) + len(pivotList))
            Invariant(Implies(len(Previous(i)) > 0, len(pivotList) > 0))
            Invariant(Acc(list_pred(arr), 1/2) and len(arr) > 0 and arr[0] == pivot)
            Invariant(MustTerminate(len(arr) - len(Previous(i))))
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more
```

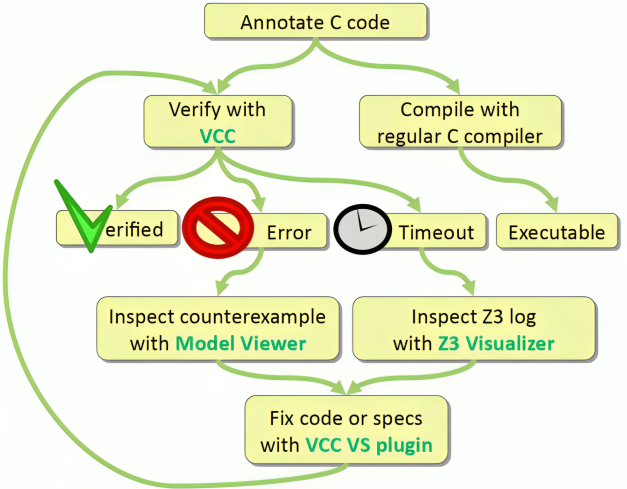
Nagini – Další informace

- Nagini
- Web frameworku Viper

VCC – Verifier for Concurrent C

- Verifikátor jazyka C s podporou:
  - Vláken
  - Vlastnictví
  - Typované paměti
- Vyvinut Microsoftem ve skupině RiSE s cílem ověřit korektnost (ve smyslu přístupu k datovým strukturám z různých vláken a dalších vlastností) jádra hypervisoru Microsoft Hyper-V
- Používá opět anotace, které obsahují z velké části i predikáty ohledně vlastnictví objektů a skupin objektů (stromů objektů)
  - Aby bylo možné verifikovat korektnost přístupu k datům z různých vláken
  - Může odhalit případy, kdy není nutné datové struktury zamykat (což je drahá operace) a zrychlit tak výsledný kód

VCC – Workflow



Zdroj: https://www.microsoft.com/en-us/research/uploads/prod/2016/02/vcc-vcc-msrc-2008-full.pptx

VCC – Příklad

```
struct LOCK {
    volatile int locked;
    spec( obj_t obj; )
    invariant( locked == 0 ==> obj->owner == this )
};
int TryAcquire(LOCK *l spec(claim_t c))
    requires(wrapped(c) && claims(c, closed(l)))
    ensures(result == 0 ==> wrapped(l->obj))
{
    int res, *ptr = &l->locked;
    atomic(l, c) {
        res = InterlockedCmpXchg(ptr, 0, 1);
        // inline: res = *ptr; if (res == 0) *ptr = 1;
        if (res) l->obj->owner = me;
    }
    return res;
}
```

VCC – Verifier for Concurrent C

- Napsat anotace tak, aby pasovaly dohromady, je velmi těžké
- Kvůli podpoře více vláken a objektů na haldě je problém jak teoreticky tak prakticky složitý
  - Proto VCC obsahuje konstrukt „assume“, kterým můžeme požadované vlastnosti funkcí deklarovat bez verifikace
- Jazyk kontraktů je zde silný, a tedy i jeho rozhodování složité
  - SMT solver (Z3) občas nedokáže rozhodnout o platnosti formule

VCC – Další informace

- Stránka projektu
- Algoritmy a principy do jisté míry publikovány ve vědeckých článcích

Dafny

- Dafny je programovací jazyk s podporou verifikace pomocí kontraktů
- Umožňuje programovat kód, který je následně ověřen vůči specifikaci
- Umožňuje *přeložit* kód do několika běžných programovacích jazyků:
  - C#
  - Java
  - JavaScript
  - Python
  - Go

DAFNY – QUICKSORT I.

```
method qsort(a: array<int>, l: nat, u: nat)
requires a.Length > 1
requires 0 <= l <= u <= a.Length
requires partitioned(a, 0, l, l, u)
requires partitioned(a, l, u, u, a.Length)

ensures sorted(a, l, u)
ensures forall i: nat :: 0 <= i < l ==> a[i] == old(a[i])
ensures forall i: nat :: u <= i < a.Length ==> a[i] == old(a[i])
ensures partitioned(a, 0, l, l, u)
ensures partitioned(a, l, u, u, a.Length)

decreases u - l
modifies a
{
        if l >= u - 1 {
        } else {
                var p_pivot := partition(a, l, u);
                qsort(a, l, p_pivot);
                qsort(a, p_pivot + 1, u);
        }
}
```

DAFNY – QUICKSORT II.

```
predicate partitioned(a: array<int>, k: nat, l: nat, m: nat, n: nat)
reads a
{
        forall i: nat, j: nat :: k <= i < l <= m <= j < n <= a.Length ==> a[i] <= a[j]
}

predicate sorted(a: array<int>, l: nat, u: nat)
reads a
requires 0 <= l <= u <= a.Length
{
        forall j, k :: l <= j <= k < u ==> a[j] <= a[k]
}

predicate eq(a: array<int>, b: array<int>, l: nat, u: nat)
reads a
reads b
requires 0 <= l <= u <= a.Length
requires a.Length == b.Length
{
        forall i: nat :: l <= i < u ==> a[i] == b[i]
}
```

Dafny – Použití

Při psaní kódu lze verifikovat kontrakty, což můžeme nahlížet jako formu ladění během vývoje:

- `dafny verify qsort.dfy`

Když je kód dokončený a verifikovaný, můžeme ho přeložit do cílového jazyka:

- `dafny build --target=py --no-verify qsort.dfy`

A následně v cílovém jazyku spustit:

- `python qsort.py`

Pro vývoj lze použít plugin pro VSCode.

Další metody pro verifikaci programů

- Verifikace využívající kontrakty je jen jednou z možností
- V poslední době je výzkum soustředěn na vývoj automatických metod, tj. metod nevyžadujících dodatečnou specifikaci
  - Vlastnosti, které je možné ověřit, jsou ale pak tímto omezeny
- Vlastnosti jsou pak specifikovány implicitně a obecně
  - Např. absence deadlocků, nezachycených výjimek, přístupů k neinicializované paměti, …
- Ověřování může probíhat jak staticky (za překladu), tak za běhu programu (kontrola vlastností v konkrétní situaci)
- Přístupy se liší i tím, zda se snaží postihnout všechny chyby
  - Typicky s rizikem hlášení neexistujících (spurious) chyb
- … nebo hlásit jen skutečně existující chyby
  - Typicky s rizikem nepostižení všech chyb

Další metody pro verifikaci programů

- Model checking kódu (nástroje: Java PathFinder, BMC, CPA, BLAST, …)
  - Pracuje nad stavy programu
- Statická analýza (nástroje: FXCop, VisualStudio, Coverity, Eclipse, IntelliJ IDEA, …)
  - Pracuje nad control flow programu
  - Nějaká forma statické analýzy je nutná pro syntax highlighting a code-completion
  - Může být velmi jednoduchá, ale dá se zkonstruovat i tak, aby vlastnosti skutečně verifikovala
- Systematické generování pokrývajících unit testů
- …

Další kurzy na MFF – Navazující magisterské studium

- NSWI101: Modely a verifikace chování systémů (ZS)
  - Základní principy a algoritmy model checkingu
  - Modelování chování systémů a jejich následná verifikace
- NSWI132: Analýza programů a verifikace kódu (LS)
  - Zaměřen na praktickou zkušenost s nástroji pro verifikaci kódu (C/C++, Java, C#)
  - Principy a algoritmy pro analýzu zdrojového kódu
- NAIL094: Rozhodovací procedury a verifikace (LS)
  - Zaměřen na SAT a SMT solvery, algoritmy, optimalizace
- **Bakalářské a diplomové práce, softwarové a výzkumné projekty**
  - Pokud Vás téma verifikací zaujalo, neváhejte mě kontaktovat!