Implementace binární řídké matice v C#

Jan Martinovič

Katedra informatiky, FEI, VŠB –TU Ostrava,
17. listopadu 15, 708 33, Ostrava-Poruba, ČR
jan.martinovic@vsb.cz

Abstrakt

Pro vytváření rychlých aplikací, vyžadujících vysoký výkon, se v současné době používá programovací jazyk C++. V tomto příspěvku popíšeme postup implementace binární řídké matice v dnes stále se rozšiřujícím jazyce C#. Zmíníme výhody a nevýhody aplikací běžících na .NET Framework run-time. Popíšeme postupy profilování těchto aplikací, které nám umožňuje nalézt slabá místa programu. Vše bude ukázáno na konkrétní implementaci binární řídké matice, realizované pomocí konečného automatu.

Klíčová slova: profilování aplikací, C#, .NET Framework, výkon aplikací

1. Úvod

Pro vytváření rychlých aplikací, vyžadujících vysoký výkon, se v současné době používá programovací jazyk C++. S rozvojem informatiky přicházejí nové programovací jazyky a technologie, jakými jsou například Java nebo .NET Framework [2,10,5] a s ním spojené programovací jazyky C#, VB.NET či manažované C++ [6]. Dále se také rozvíjí využívání technik, jakými jsou refaktoring [4] nebo extrémní programování [1]. V [4] se můžeme také dočíst o problémech při hledání slabin v kódu a o jejich odstranění pomocí profilování.

V tomto článku popíšeme postup implementace binární řídké matice v stále více se rozšiřujícím programovacím jazyce C#. Popis nebude zaměřen přímo na kód naší implementace, ale popíšeme problémy, se kterými jsme se při implementaci setkali a jejich příklady řešení pomocí různých profilovacích nástrojů.

Článek je rozdělen do tří částí. V první je uveden nástin problematiky reprezentace binární řídké matice pomoci konečného automatu. Následující část obsahuje popis tří nástrojů pro vylepšení funkčnosti a výkonu aplikací. Na závěr jsou uvedeny výhody a nevýhody aplikací psaných v řízeném kódu.

2. Binární řídká matice reprezentovaná konečným automatem

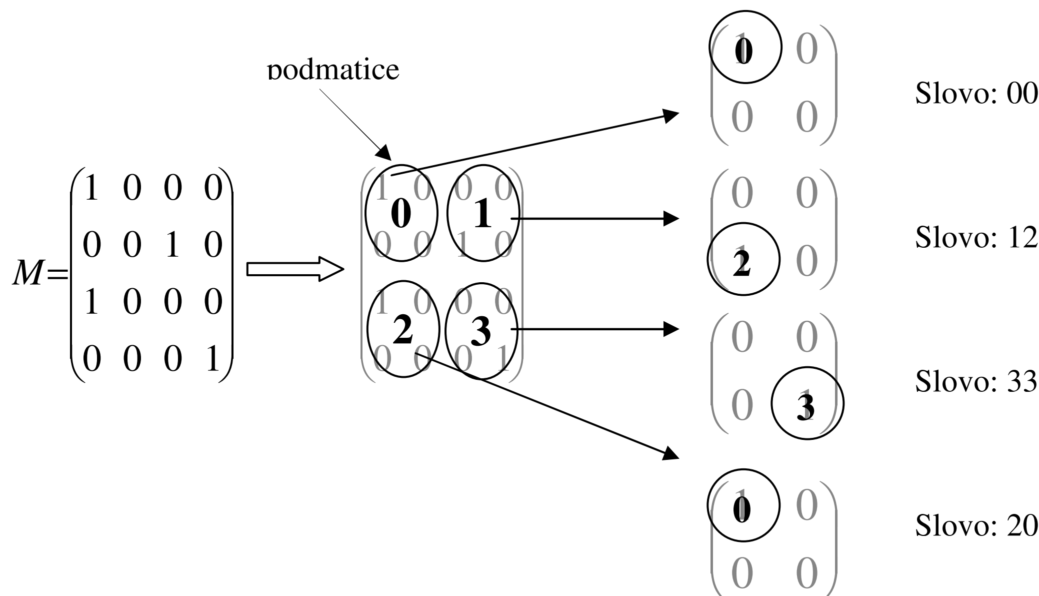
V experimentech které provádíme, často pracujeme s řídkými maticemi velkých rozměrů. Proto urychlení práce s těmito maticemi se používají různé metody ukládání dat. My jsme se zabývali implementací binární řídké matice, která by byla po celou dobu práce s ní uložena v operační paměti počítače a měla by stejně rychlý přístup k sloupcům i řádkům.

Metoda, kterou používáme je založena na reprezentaci binární řídké matice pomocí konečného automatu a dělení větších matic na 4 menší podmatice [3]. Dělení na podmatice se provádí, dokud není velikost nové podmatice rovna 1.

Větší matici v automatu reprezentuje stav a menší matice jsou reprezentovány stavem a přechodem, se kterým se k těmto maticím dostaneme:

$$M_1 = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} ,$$

Mějme matici M obsahující m sloupců a n řádků. Automat je definován abecedou: $\Sigma = \{0,1,2,3\}$. Slova přijímaná automatem jsou délky l a platí: $k = 2^l$, kde $k \geq n$ a zároveň $k \geq m$.
Pro lepší představu o konstrukci automatu si uvedeme jednoduchý příklad:



kde M je binární řídká matice, kterou chceme reprezentovat pomocí konečného automatu. Do automatu se ukládají pouze nenulové body matice. Slova která bude přijímat náš automat budou tato: $L = \{00,12,20,33\}$.

3. Profilování aplikací napsaných v .NET Frameworku

V následujících sekcích popíšeme programy, které lze použít k profilování aplikací napsaných nad řízeným kódem. Programy, které jsou zde popsány, fungují jak pro aplikace napsané v .NET Frameworku 1.1, tak pro aplikace napsané v .NET Frameworku 2.0 beta 2.

V každé sekci si v krátkosti popíšeme příslušný program a poté si uvedeme příklad jeho použití. Budeme popisovat příklady, vycházející z testů, které jsme používali při psaní výše uvedené binární řídké matice.

3.1. CLR Profiler

CLRProfiler [7] nám umožňuje analyzovat chování aplikace napsané v řízeném kódu. Jako jiné profilovací aplikace má svoje silné stránky a své slabiny. Proto by neměl být pouze jediným profilovacím nástrojem, který používáme, ale součástí balíků profilovacích nástrojů. CLRProfiler patří mezi nástroje, které se soustředí na analýzu toho, co se děje s pamětí spravovanou pomocí 'garbage collectoru' (tzv. halda).

Pomáhá nám odpovědět na následující otázky:

- Které metody alokují objekty příslušných typů?
- Které objekty zůstávají v paměti a jak dlouho?
- Co je umístěno v paměti?
- Co udržuje objekty v paměti?

Dále nám umožňuje zobrazit:

- Graf ukazující kým byl objekt volán a jak často.
- Kým byly které metody, třídy a moduly spuštěny.

CLRProfiler nám umožňuje profilovat lokální aplikace, služby a také ASP.NET aplikace. Výsledky profilování se ukládají do logovacích souborů, které nám pomáhají při srovnávání jednotlivých verzí programů a případném pozdějším zkoumání výsledků.

Příklad použití: Nyní si popíšeme využití aplikace CLRProfiler k zvýšení výkonnosti. Příklad bude popsán na jednom z testů prováděných nad naší binární řídkou maticí. Mezi výhody, které nám tyto testy poskytují, patří nalezení vhodných metod pro realizaci složitých výpočetních operací. Pokud se při psaní dalšího kódu budeme těmito poznatky držet budou aplikace, které budeme psát výkonnější.

Na obrázku 1 můžeme vidět, kolik paměti využívá naše aplikace jako celek a kolik paměti využívají jednotlivé objekty. Zde je nutno upozornit, že se jedná o paměť, které je spravovaná pomocí 'garbage collectoru' - halda. Neobjeví se zde paměť alokovaná na zásobníku - ta je pod plnou kontrolou a správou programátora.

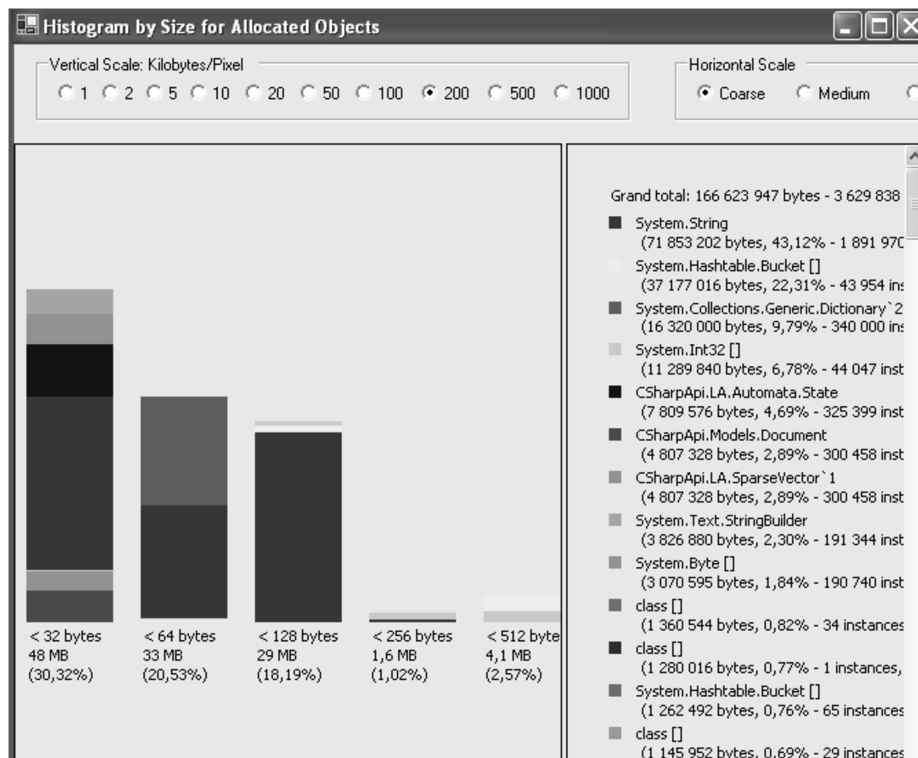
Nyní se vraťme k našemu obrázku. Na obrázku je patrné, že nejvíce paměti je využíváno třídou System.String. V průběhu činnosti programu bylo vytvořeno 1.891.970 objektů této třídy. V předchozích testech jsme tříd typu System.String měli méně než jeden milión.

Poté co jsme zjistili tento velký nárůst objektů třídy System.String, začali jsme zkoumat nově přidaný kód. Fragment z tohoto kódu si nyní uvedeme:

```
1: List<uint> resultList = new List<uint>();
2: for (uint s = 1; s < baseSize; s = s * 2 )
        // kde baseSize = Math.Pow(2,19);
3:  {
4:     …
5:    for (int i = 0; i < resultList.Count; i++)
6:     {
7:       for (byte j = 1; j < 5; j++)
8:       {
9:          uint position = Get(resultList [i], j);
10:         if (position != 0)
11:         {
12:             StringBuilder result = new StringBuilder;
13:             MemoryPath(position, result);
                //v metodě dochází k max. 17 přičtením znaků
14:             string hash = result.ToString();
                // převod na string kvůli dalším výpočtům
15:                 …
16:            }
17:         }
18:       }
19:  }
```

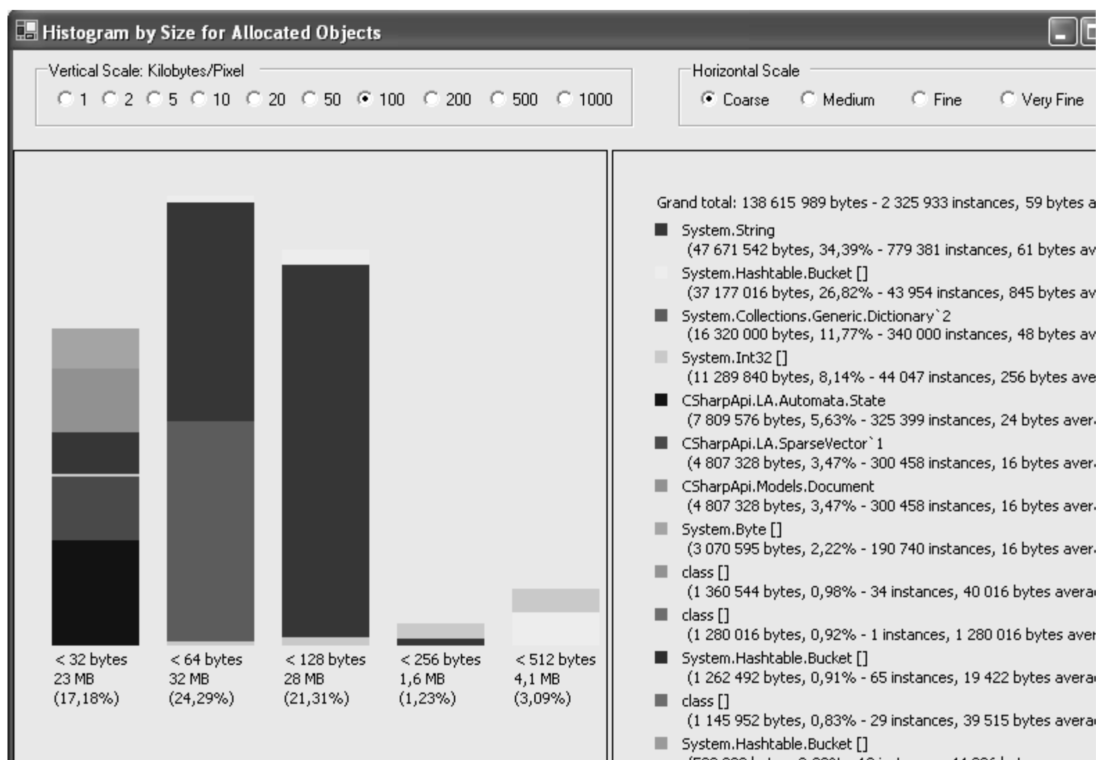
Ve výše uvedeném kódu nás zajímají řádky 12 – 14, ve kterých se vytváří objekt třídy StringBuilder. Poté se naplní maximálně 17-ti znaky a převede na klasický String. Převod na String je z důvodu rychlé funkce Hashtable při vyhledávání řetězců (další možností by bylo napsání speciální třídy Hashtable, která by prováděla rychlé vyhledávání v objektech třídy StringBuilder). Úsporu paměti a urychlení tohoto kódu jsme docílili změnou třídy StringBuilder na pole znaků, které se vytvoří pouze jednou. K jeho naplnění a nulování dochází v metodě MemoryPath. Fragment upraveného kódu je uveden zde:

```
0:  char[] result = new char[17];
…
12:       // smazán
13:       MemoryPath(position, result);
      // v metodě dochází k max. 17 přičtením nových znaků
14:   string hash = new String(result);
```

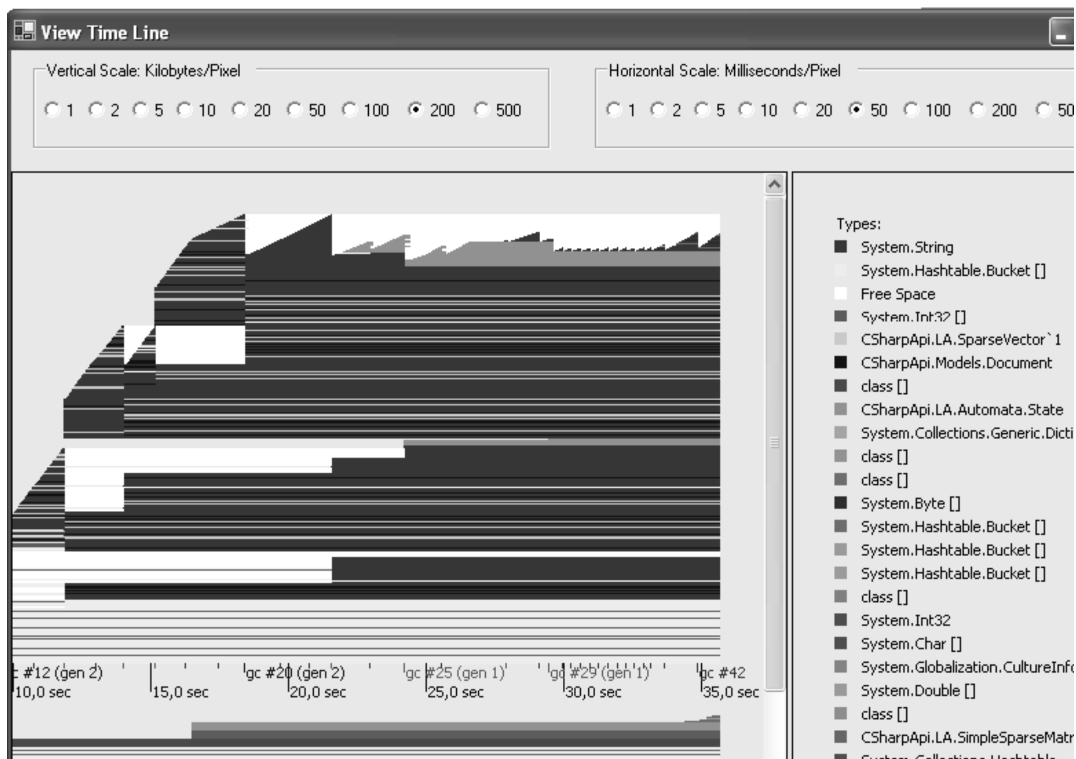


Obr. 1. Příklad výsledku aplikace CLRProfiler

Na obrázku 2 můžeme vidět výsledek profilování po provedených úpravách kódu.



Obr. 2. Příklad výsledku aplikace CLRProfiler - po úpravě



Obr.3. Příklad časové osy

3.2. NProf

Další z profilovacích nástrojů, které si představíme se jmenuje NProf [9]. NProf podle svých autorů není pouze nástroj pro profilování .NET aplikací, ale obsahuje také kompletní API pro vytváření jiných profilovacích aplikací. Další možností je možnost rozšíření původního GUI o nové vizualizační prvky. Architekturu tohoto profileru můžeme rozdělit do následujících úrovní:

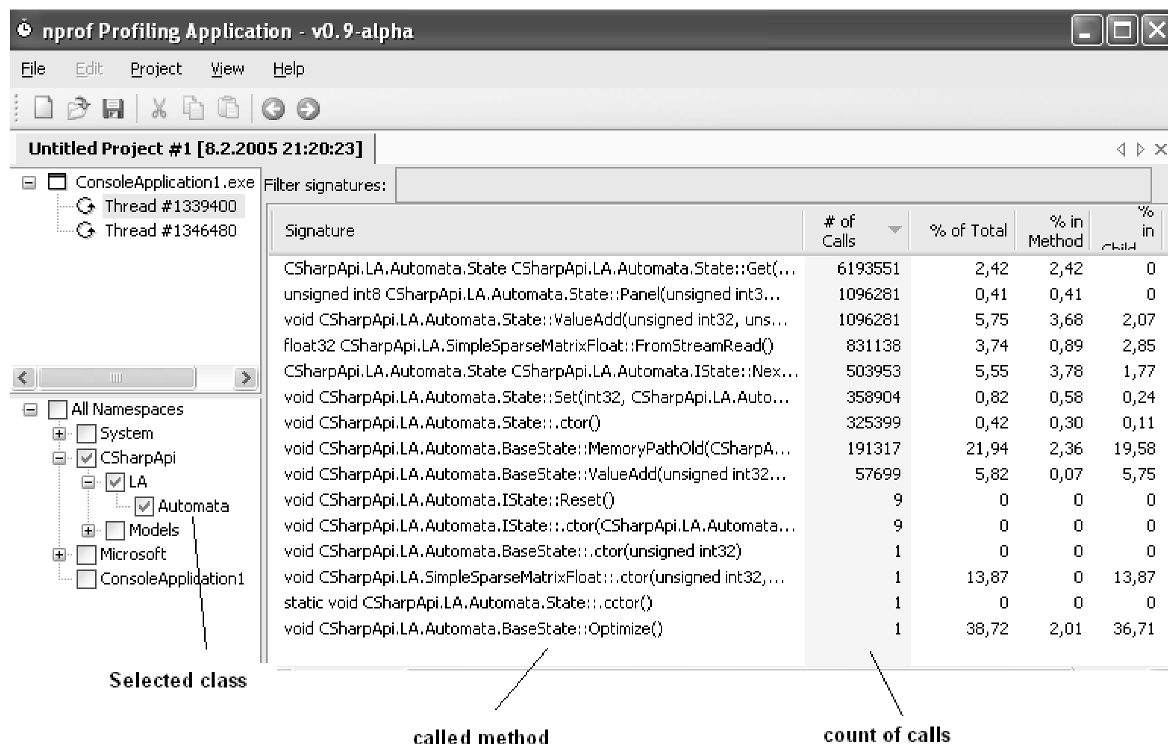
Vrstva 1: Obsahuje C++ knihovnu NProf.Hook.dll, která se připojuje k cílovému procesu pomocí rozhraní COR profilovacích proměnných. Tato .dll knihovna shromažďuje informace, seskupuje je a pomocí lokálního soketu odesílá do vrstvy 2. Přístup pomocí soketu je v tomto profilovacím nástroji použit z důvodu, že není nutno provádět další externí konfigurace, jak je to u profilerů používajících pojmenované roury, které potřebují některé další bezpečnostní nastavení konfigurace pro profilování ASP.NET aplikací. Tento přístup také podle autorů profileru umožňuje jednoduchou úpravu pro vzdálené profilování. Zajímavosti této úrovně je, že jako jediná obsahuje neřízený kód.

Vrstva 2: Tato vrstva obsahuje řízený kolektor ProfilerSocketServer. Vrstva poslouchá na soketu, do kterého vrstva 1 zapisuje. Shromažďuje data a ukládá je do řízených kolekcí a objektů. Tato vrstva může být také odpovědná za nasazování určitých COR událostí pro 'real-time' profilování.

Vrstva 3: Tato vrstva obsahuje třídy grafického uživatelského rozhraní a to v jmenném prostoru NProf.GUI. Třídy GUI jsou odpovědné za zobrazování informací z vrstvy 2. GUI třídy jsou vestavěné do čtvrté vrstvy, aby pomohly vytvářet kompletní aplikaci. Důvod pro vytvoření speciální vrstvy pro tyto třídy je, že je lze použít jak v aplikacích běžících samostatně, tak jako aplikacích zabudovaných přímo do VS.NET.

Vrstva 4: Třídy samotné aplikace (NProf.VSNetAddin and NProf.Application) se nacházejí v této vrstvě. Nachází se zde jednoduchý hostitel pro GUI třídy ze třetí vrstvy, právě tak jako správce informací o projektech uložených v třídách - NProf.Glue.Profiler.Project.

Příklad použití: Příklad použití aplikace NProf je uveden na obrázku 4. Je na něm vidět počet volání jednotlivých metod obsažených v implementaci výše uvedené binární řídké matice.



Obr. 4. Příklad výstupu aplikace NProf

3.3. FxCop

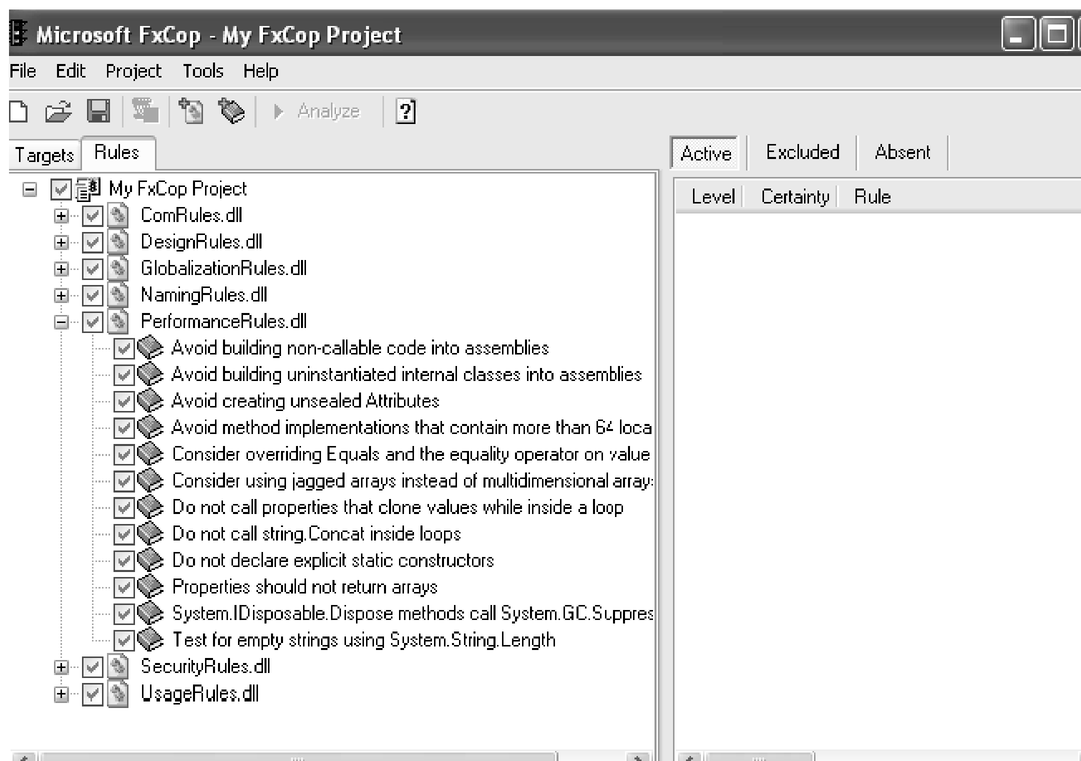
Poslední aplikaci, kterou si uvedeme je FxCop [8]. Neprovádí profilování běžící aplike, ale analyzuje kód řízené assembly a vrací informace o této assembly. Informace, které můžeme pomocí tohoto nástroje získat jsou: jestli jsme použili dobrý návrh, zda správně používáme lokalizace, pomůže najít kód zhoršující výkon aplikací a může nám poskytnou informace pro vylepšení bezpečnosti našich aplikací.

FxCop je určen především pro vývojáře knihoven. Ale může být používám všemi, kteří chtějí vytvářet aplikace přizpůsobené .NET Frameworku. Další kategorii uživatelů mohou být studenti, kteří se chtějí seznámit s .NET Frameworkem a například s korektním psaním „štábní kultury“ doporučované autory aplikace FxCop.

FxCop je vytvořen tak, aby mohl být začleněn do celého cyklu vývoje softwaru.

Příklad použití: Při použití aplikace FxCop jsme byli schopni jednoduše kontrolovat, zda jsme správně dodržovali jmenné konvence. Mezi zajímavá pravidla, která FxCop obsahuje patří například následující:

- Pro testování prázdných řetězců otestujte zda String.Length je rovno nule.
- Nepoužívejte string.Concat uprostřed cyklu - používejte místo něho StringBuilder.



Obr. 5. Příklad části výstupu aplikace FxCop

4. Závěr

V článku jsme se snažili ukázat význam používání profilovacích nástrojů při tvorbě aplikací. Výhodou řízeného kódu je rozrůstající se množství nástrojů pro profilování. Jistou nevýhodou je ztráta výkonu řízeného kódu oproti kódu napsaného v C++. Zde je ale potřeba upozornit, že ne každý kód napsaný v neřízeném C++ musí být vždy rychlejší než kód řízený [2]. O výkonu aplikací nerozhoduje tedy jen programovací jazyk, který pro jejich tvorbu použijeme, ale i způsob s jakým využíváme výhod daného programovacího jazyku. Nedílnou součástí je také volba vhodného výpočetního algoritmu.

Literatura:

1. Beck K.: Extrémní programování, Grada 2002, ISBN 80-247-0300-9.
2. Burton K.: .NET Common Language Runtime Unleashed, by Sams Publishing, 2002, ISBN: 0-672-32124-6.
3. Dvorský J., Krátký M.: Mutli-dimensional Sparse Matrix Storage, DATESO 2004.
4. Fowler M. a kol: Refaktoring - Zlepšení existujícího kódu, Grada 2003, ISBN 80-247-0299-1.
5. Mariani R.: Garbage Collector Basics and Performance Hints, Microsoft Corporation, April 2003,
   http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetGCbasics.asp
6. C++/CLI Community: http://blog.cpp-cli.org.
7. CLR Profiler (v2.0): http://www.microsoft.com/downloads.
8. FxCop Team Page: http://www.gotdotnet.com/team/fxcop.
9. nprof the .NET profiler: http://nprof.sourceforge.net/Site/SiteHomeNews.html.
10. Microsoft NET Framework Developer Center: http://msdn.microsoft.com/netframework.

Anotation:

The C++ programming language is largely used for making the applications which need height performance. There are described techniques of implementation the binary sparse matrix based on finite automaton. We used the still growing up C# language for our implementation. We will describe advantages and disadvantages of the applications which use the .NET Framework run-time. All of this will be demonstrated on our implementation of the binary sparse matrix.