УДК 004.42 (042.3)

Е. М. СИДОРОВ

Національний авіаційний університет, м. Київ

ЗВОРОТНЯ ШНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВІАЦИЙНИХ ТРЕНАЖЕРІВ

Аннотация. Рассматривается метод обратной инженерии наследуемого программного обеспечения авиационных тренажеров. Описывается способ построения декомпилятора как части средств обратной инженерии, обеспечивающих применение рассмотренного метода, рассматривается состав декомпилятора и особенности процесса декомпиляции.

Ключевые слова: унаследованное программное обеспечение, реинженерии, обратная инженерия, декомпиляция, авиационные тренажеры

Анотація. Розглянуто метод зворотної реінженерії успадкованого програмного забезпечення авіаційних тренажерів. Описується спосіб побудови декомпілятора як частини засобів зворотної інженерії, що забезпечують застосування розглянутого методу, розглядається склад декомпілятора і особливості процесу декомпіляції.

Ключові слова: успадковане програмне забезпечення, реінженерія, зворотня інженерія, декомпіляція, авіаційні тренажери

Abstract. The reverse engineering approach of legacy software of aviation simulator is presented. The method of realizing decompile as part of the reverse engineering tool based on the reengineering approach is looked. Using the reverse engineering approach on example of legacy simulator software is presented.

Keywords: inherited software, reengineering, reverse engineering, decompiling, flight simulators

Вступ

Вже багато років в авіаційній техніці застосовуються цифрові обчислювальні системи, невід’ємною складовою яких є програмне забезпечення (ПЗ). До цієї техніки відносяться авіаційні тренажери, найкоштовні складові навчально-тренувальних комплексів. При їх експлуатації виникає задача підтримки придатності ПЗ. Ця задача постає в двох випадках, по-перше, коли здійснюється доробка, модифікація або відновлення авіаційних тренажерів, по-друге, коли створюються їх нові покоління [1]. В першому випадку, при вирішенні задачі підтримки придатності ПЗ здійснюється часткова або повна його переробка, в другому випадку, ПЗ створюється заново, але обов’язково з урахуванням попереднього досвіду. В обох випадках доцільно застосовувати реінженерію ПЗ.

Актуальність

Існуючі методи реінженерії ПЗ для перевірки результатів використовують порівняння, або результатів виконання успадкованого і заново побудованого ПЗ, або результатів реверсивної інженерії з поведінкою відповідної моделі домена [2]. Але бувають ситуації, коли ці методи неможливо застосовувати, тому що, по-перше, внаслідок заміни відсутнє обчислювальне обладнання, на якому можна виконати успадковане ПЗ, а по-друге, опис моделі домена, яку представляють в документації, як правило містить помилки. При цьому, вирішальну роль в обґрунтуванні адекватності функціонування розробленого ПЗ функціонуванню реального об'єкту, грають властивості цього об'єкту. Тому, розробка та дослідження відповідного метода реінженерії ПЗ авіаційних тренажерів є актуальною.

Метою роботи є розробка метода реінженерії ПЗ авіаційних тренажерів, який застосовується при заміні обчислювального обладнання тренажеру та відсутності точного опису математичної моделі відповідного домену в документації, і який враховує відомі властивості об’єкту. В статті розглянуто запропонований метод та окремі засоби, щодо його реалізації, а саме асоби декомпіляції об’єктного коду.

Постановка задачі

Виконання процесів супроводження ПЗ призвело до необхідності реконструювання програм та розробки відповідного розділу інженерії ПЗ, який називається зворотною (backward) або реверсивною (reverse) інженерією [3]. Підхід, який забезпечує процеси розробки ПЗ називається прямою (forward) інженерією. Задача зворотної інженерії протилежна задачі прямої інженерії та полягає в забезпеченні процесів отримання з низькорівневого представлення ПЗ (похідного коду), його високорівне-вого представлення, наприклад, проектної інформації, або специфікацій вимог. Тому, основна мета зворотної інженерії – відновлення інформації про ПЗ, а підставою для формулювання цієї мети є необхідність розуміння успадкованого ПЗ. Так як, ПЗ майже завжди представлено похідним кодом, то усі відомі методи та засоби відновлення інформації про ПЗ орієнтовані саме на похідний код. В розділі наведено результати аналізу методів та засобів розуміння ПЗ. Реінженерія (reengineering) ПЗ – це підхід, що забезпечує переробку ПЗ, ґрунтуючись на досвіді відповідного домену, та будується шляхом взаємозв’язку зворотної та прямої інженерій («кругообіг» ПЗ) (рис. 1).

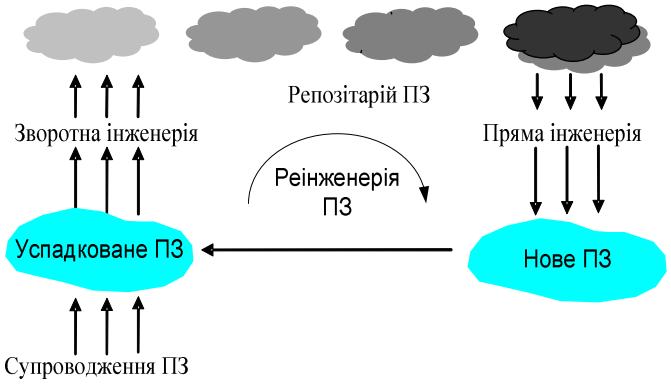

Рисунок 1 – «Кругообіг» програмного забезпечення

Особливості домену та місця тренажеру в ньому визначають те, що реінженерія успадкованого ПЗ цього типу окрім вирішення традиційних завдань, пов'язаних з відновленням проектної інформації, вимагає відновлення інформації про реальний об'єкт та особливого підходу до рішення задачі обгрунтування адекватності функціонування побудованого в результаті реінженерії ПЗ поведінці реального об'єкту. Інформація про реальний об'єкт, відновлювана в процесі реверсивної інженерії, – це алгоритми, моделі, характеристики вхідних параметрів, датчиків, приладів та виконавчих пристроїв або моделі реального об'єкту. При відновлюванні, окрім шляху для отримання інформації – аналізу успадкованого коду та документації – слід використовувати експериментальне дослідження поведінки моделі. Обгрунтування адекватності функціонування побудованого ПЗ не може здійснюватися відомими методами – порівнянням результатів виконання успадкованого та нового ПЗ або порівнянням результатів реверсивної інженерії з поведінкою відповідної моделі домену. Перший метод не можна використовувати за відсутністю, внаслідок заміни обчислювального обладнання, на якому можна виконати успадковане ПЗ, а другий, – оскільки модель домену, представлена в документації, як правило, містить помилки. Окрім цього, особливе положення ПЗ в домені указує на те, що основна увага при обгрунтуванні адекватності повинна приділятися детальному аналізу його функціонування в реальних умовах. Все це свідчить про те, що вирішальну роль в обгрунтуванні адекватності функціонування розробленого в результаті реінженерії ПЗ, функціонуванню реального об'єкта повинні грати характеристики та властивості реального об'єкту. Це складає сутність запропонованого методу керованої об’єктом реінженерії ПЗ (рис. 2.).

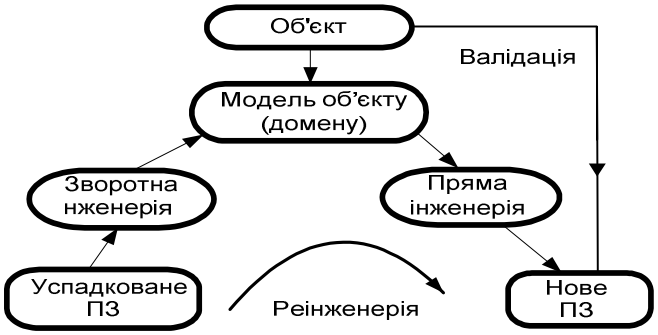

Рисунок 2 – Схема методу ре інженерії

Для аналізу похідного коду (мова програмування SYPS платформи Robotron), та побудови його високорівневого алгоритмічного уявлення на платформі Microsoft.Net та мові програмування C# було побудовано інструмент зворотної інженерії (за архітектурою «екстрактор - абстрактор» [4]). Він має графічний інтерфейс користувача і є Web-застосуванням. Послідовність функцій, яку виконує інструмент наступна: завантаження похідного коду для виконання аналізу; читання коду; перевірка помилок; аналіз похідного коду (екстрактор); трансляція в мову C# (екстрактор); запис в .dot файл; завантаження файла; виклик графічного модуля (абстрактора). Однією з найважливіших дій, яка забезпечує аналіз є

декомпіляція успадкованого об’єктного коду. Тому, декомпілятор є складовою частиною екстрактора. Від якості його функціонування залежить якість зворотньої інженерії в цілому.

Декомпілятор

Декомпіля́ція - процес відтворення вихідного коду декомпілятором. Декомпіля́тор - це програма, що транслює текст, отриманий на выході компілятора, у відносно эквівалентний вихідний код на мові програмування високого рівня. Дизассемблер, як окремий випадок декомпілятору транслює текст програми в текст на мові ассемблера. Результативність декомпіляції залежить від об'єму інформації, представленої у коді, що декомпілюється. Декомпілятор має структуру схожу на структуру компілятора. Типові складові такі: синтаксичний аналізатор, семантичний аналізатор, генератор проміжного коду, генератор графа потоків керування, аналізатор потоків даних, аналізатор потоків керування, генератор коду. На практиці деякі із цих фаз опускаються або поєднуються в одну.

Синтаксичний аналізатор, групує байти вихідної програми в граматичні фрази (або речення) вихідної машинної мови. Ці фрази можна представити у вигляді простих виразів присвоювання, наприклад, (ecx := ecx - 50) або переходу (jmp 0x401000). Основна задача, що вирішується синтаксичним аналізатором у складі декомпілятора - визначення, що є кодом, а що даними. Наприклад, таблиця переходів оператора case може бути розташована в сегменті коду, і декомпілятор не може відрізнити її від інструкцій через особливості архітектури X86. Тому не можна проводити просто послідовний аналіз, припускаючи, що наступна послідовність байт завжди містить інструкцію. Для розв'язку потрібні машинно-залежні евристики.

Семантичний аналізатор вихідної програми визначає семантичне значення груп інструкцій, здійснює збір інформації про типи й поширення цих типів усередині підпрограм. Для перевірки семантичного значення групи інструкцій використовується набір ідіом. Так, наприклад, інструкція: | shl eax,2, може бути перетворена в | mul eax,4. Для подальшого аналізу програми декомпілятор перетворює програму в проміжне представлення. Це представлення повинно легко генеруватися з вихідної програми, а також воно має бути підходящим для генерації надалі результуючої програми. Для платформи i386 добре підходить трьохадресне представлення (приймач, джерело_1, джерело_2), тому що всі інструкції мають максимум три операнда. Надалі ці операнды можна легко розширити до виразів, що представляють високорівневі конструкції.

Генератор проміжного коду будує проміжний коду, на якому можна проводити оптимізацію, а також створювати компілятори, що працюють на кількості платформ і розуміють кілько мов [5].

Генератор графу потоків керування створює граф для аналізу програми, який представляє потокі керування кожної підпрограми, що необхідне для виділення високорівневих конструкцій (умов, циклів), використовуваних у програмі. Граф також допомагає видалити проміжні інструкції начебто безумовних переходів, що вводяться через існування обмеження на діапазон адреси в умовному переході.

Аналізатор потоків даних оптимізує проміжний код, шляхом визначення високорівневих виразів. Усувається визначення й використання проміжних регістрів, прапорів умов, тому що ці поняття не існують у високорівневих мовах.

Аналізатор потоків керування структурує граф потоків керування кожної підпрограми, шляхом виділення множини узагальнених високорівневих конструкцій. Ця узагальнена множина повинна містити керуючі інструкції, використовувані в більшості мов, такі як цикли й умови. Однак не повинні включатися конструкції, специфічні, тому що бажано забезпечити як можна більшу незалежність від конкретної мови.

Генератор коду створює код потрібною мовою високого рівня, шляхом використання графа потоків керування й проміжного коду кожної інструкції. Для кожної реєстрової й стековой змінної, аргументів і самих процедур призначаються імена. Структури керування й проміжні інструкції перетворюються у високорівневі вирази.

В статті декомпілятор містить наступні складові: front-end, універсальний і аналізатор ідіом, back-end.

Front-end виконує процес, який складається з фаз, залежних від ЕОМ або від машинної мови. Ці фази включають лексичний, синтаксичний і семантичний аналіз, генерацію проміжного коду й графа потоків керування. Результатом є проміжне, машинно-незалежне представлення програми.

Універсальний аналізатор є незалежним, як від машини, так і від мови, на якій потрібно одержати текст програми. Він є ядром декомпілятора й робить найбільш важливі перетворення, від якості яких в основному й залежить якість декомпілятора. Він реалізує процес, що складається з аналізу потоків даних і аналізу потоків керування.

Аналізатор ідіом виконує аналіз проміжного коду з метою пошуку в ньому специфічних конструкцій, які не є загальними для всіх мов. Однак цей етап не може бути перенесений у фазу back-end, тому що після аналізу ідіом можуть продовжуватись інші види універсального аналізу. Прикладом таких ідіом є викликі віртуальних функцій, після яких продовжуватись аналіз потоків даних.

Back-end генерує текст програми. У теорії компіляторів подібне групування фаз використовується розробниками для створення компіляторів для різних вихідних мов і різних машин. Тому, якщо переписати back-end компілятора для нової платформи, можна використовувати існуючий front-end. На практиці існують обмеження у використанні такого підходу, які головним чином пов'язані з вибором виду проміжного коду.

Таким чином, завдяки запропонованому способу створення конструкції декомпілятора, шляхом заміни front-end і back-end частин можна одержувати декомпілятори, які здатні "читати" програми на різних машинних мовах і "писати" їх представлення на різних мовах високого рівня. Розглянемо деякі особливості декомпіляції, які було запропоновано при реалізації методу реінженерії. Програма складається з даних і інструкцій з їхньої обробки: Нехай: P – програма, I = {I1, I2, ..., In} - інструкції P, D = {D1, D2, ..., Dn} - дані P, тоді P = I + D. Інструкції зручно розглядати групами, послідовностями. Це дозволяє зменшити розміри графів, а також абстрагуватися від окремих інструкцій, коли вони не відіграють ролі. Проміжні інструкції ділимо на дві множини:

- переходу: множина інструкцій, які передають керування на адресу, відмінну від адреси безпосередньо наступної інструкції - умовний і безумовний переходи, виклик підпрограми, повернення з підпрограми, кінець програми;
- не містять переходу.

Базисний блок - це послідовність інструкцій, яка має один вхід і один вихід. Якщо виконується одна інструкція блоку, виконується й увесь блок. Множина інструкцій програми може бути розділена на множину непересічних базисних блоків, починаючи із точки входу програми.

Граф потоків керування – це направлений граф, який представляє потоки керування в програмі. Вершинами цього графа є базисні блоки, а ребрами - передачі керування між блоками.

Низькорівневий проміжний код, що генерується front-end, подібно ассемблерному, використовує регістри й коди умов. Це представлення може бути перетворене в інше, більш високорівневе. Перетворення традиційно називають оптимізацією [5].

Типи перетворень, які проводяться на етапі аналізу потоків даних, включають:

- аналіз інструкцій у цілому, видалення пустих інструкцій (nop; xchg eax, eax), усунення дубльованих виразів, усунення спекулятивних обчислень;
- аналіз регістрів, поширення регістрів, видалення мертвих регістрів, визначення регістрів-результатів у функціях, визначення регістрових параметрів;
- аналіз прапорів умов (прапорів), поширення прапорів, видалення мертвих прапорів.

Більшість із цих перетворень поліпшують якість низькорівневого коду й дозволяють відновити частину інформації, втраченої під час процесу компіляції. Крім того, деякі із цих перетворень служать для усунення наслідків оптимізацій, зроблених компілятором, підготовляючи код для подальших перетворень більш простими алгоритмами. До перетворень пред'являється кілька вимог: перетворення повинне зберігати логіку програми; перетворення повинне бути вигідним.

Після виконання перетворень над графами потоків даних та керування внутрішнє представлення набуває структури, подібної до оригінальної структури програми. Для остаточного приведення її до виду, з якого можна за допомогою кодогенератора отримати декомпільований код, необхідно відмітити специфічні аспекти для конкретної мови програмування (або компілятора):

- розташування глобальних та локальних змінних;
- система типів, що використовуються;
- прототипи функцій;
- типи та семантика змінних;
- прототипи функцій з сторонніх модулів, на які присутні посилання.

Після передачі представлення до кодогенератора отримаємо декомпільований код, еквівалентний за функціоналом до оригінального машинного представлення.

Застосування методу реінженерії

Успадковане ПЗ характеризує дискретний процес моделювання, а необхідна швидкодія досягалася за рахунок наступного(рис. 3): спрощенням математичних моделей динаміки польоту та систем літака; заміною аналітичного опису реальних (динамічних) процесів залежностями у вигляді таблиць рішень; використанням простих, але швидких методів інтерполяції функцій; використанням простих методів інтегрування диференційних рівнянь.

На рис. 4 показана загальна схема виконаної міграції програмно-технічного забезпечення тренажера.

Оскільки, побудоване ПЗ не можна було перевірити на правильність функціонування шляхом виконання успадкованого ПЗ (обчислювач було замінено), то реверсивна інженерія, окрім традиційних процесів, включала процес додаткової взаємної перевірки успадкованого коду та моделі.

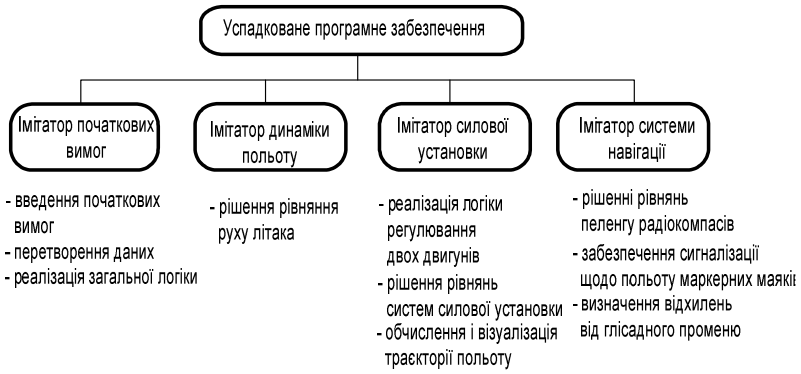

Рисунок 3 – Склад успадкованого програмного забезпечення

Більш всього помилок було в коефіцієнтах рівнянь моделі, що описує динаміку польоту. Тому для їх уточнення використовувався успадкований код, в якому шляхом декомпіляції за наведеною схемою визначалися частини, що здійснюють обчислення коефіцієнтів. Інтерпретація цих частин «за столом» дозволила визначити значення помилкових коефіцієнтів.

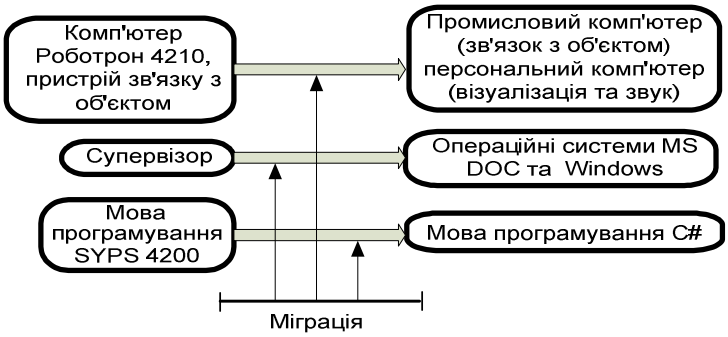

Рисунок 4 – Міграція технічного та програмного забезпечення тренажера

Реінженерія успадкованого ПЗ після перевірки моделей здійснювалася в два етапи: реверсивна інженерія – побудова моделі та алгоритмічного уявлення; пряма інженерія – по отриманих моделях та алгоритмах, будувався код нового ПЗ в мові C#. Для виконання реверсивної інженерії за відомою схемою був побудований спеціальний інструмент, який містить екстрактор та абстрактор. Робота інструмента та результат прямій інженерії на фрагменті коду показаний на рис. 5 (розглядується модель параметра – ознаки перебування двигуна в режимі розкручування ротора стартером).

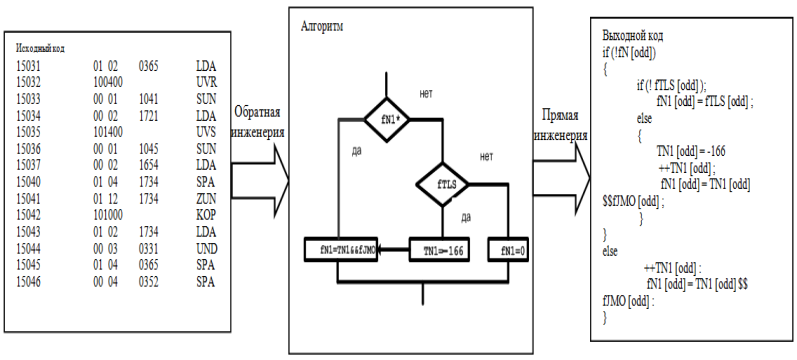

Рисунок 5 – Результат реінженерії (фрагмент)

Дослідження процесів реінженерії ПЗ, вперше дозволило виявити декілька схем їх виконання за наступним сценарієм: $S : L \xrightarrow{C} N$, де S – позначення схеми реінженерії; L – успадковане ПЗ; N – нове ПЗ; C – множина умов, від яких залежить застосування схеми. Було встановлено чотири наступні базові схеми реінженерії ПЗ: «код-код», «код-алгоритм-код», «код-модель-алгоритм-код», «код-модель-

технічний опис-алгоритм-код». Схему «код-модель-технічний опис-алгоритм-код» наведено на рис.6, в якості прикладу. Умовою (С) для застосування цієї схеми є наступне: успадкований код (L) реалізує невідому модель, створену розробником, з якої зрозуміти алгоритми неможливо; опис змісту фізичного процесу, що моделюється, є в технічній документації реального об'єкту. Сполучення моделі та технічного опису дозволяє побудувати алгоритм та створити нове ПЗ (N).

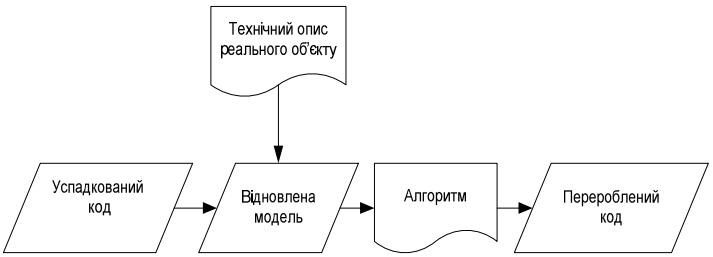

Рисунок 6 – Схема реінженерії

По цій схемі була проведена реінженерія більшості програм для моделювання параметрів імітаторів «силова установка» та «динаміка польоту», а також декількох параметрів імітатору «навігаційна система».

Висновки

Запропонований метод було перевірено при реалізації засобів реінженерії програмного забезпечення авіаційного тренажеру TL-410. Метод має наступні переваги порівняно з існуючими [6]:

- універсальність – одну реалізацію ядра декомпілятора можна використовувати для декомпіляції програм на кілька мов та кілька архітектур, для цього достатньо лише замінити front-end;
- якість результуючого коду – в результаті ітеративного спрощення графів потоку даних та керування декомпілятор генерує код, структурно дуже близький до оригіналу.

Застосування методу керованої об'єктом реінженерії в поєднанні з запропонованим методом побудови декомпілятора дозволило ефективно реалізувати реінженерію успадкованого програмного забезпечення авіаційного тренажеру TL-410.

Автор висловлює подяку студенту Старовому С., який приймав участь в реализації декомпилятора.

Література

1. Хоменко В.А. Шаблон программного обеспечения устройств связи с объектом авиационных тренажеров / Хоменко В.А., Сидоров Е.Н., Мендзебровский И.Б. // Проблеми програмування; НАН України.- 2008.- №2, -3.- С. 239-249.

2. Сидоров Н.А. Реинженерия наследуемого програмного обеспечения авиационных тренажеров / Сидоров Н.А., Хоменко В.А., Недоводеев В.Т., Сидоров Е.Н. // Проблеми програмування; НАН України. - 2008.- №2, -3.- С. 288-299.

3. Сидоров Н.А. Реинженерия наследуемого программного обеспечения информационно-моделирующих тренажерных комплексов / Сидоров Н.А., Недоводеев В.Т., Сердюк И.П., Хоменко В.А., Сидоров Е.Н. //Управляющие системы и машины.-№4.-2008.-С.68-74.

4. Chikofsky E.J., Gross J.H., Reverse Engineering and Designing Recovery Taxonomy. – IEEE Software. – January. – 1990.- p. 13-17.

5. Kaspersky K. The art of decompilation. BHV, 2009. – 378 p.

6. Clifford Roy. Decompilation of object programs. Dept. of Electrical Engineering, 1973. – 275 p.

Стаття надійшла: 30.05.2013.

Відомості про авторів

Сидоров Євген Миколайович – к.т.н., доцент кафедри комп'ютерних інформаційних технологій, Національний авіаційний університет.