УДК 004.4’24 https://doi.org/10.15407/pp2020.02-03.157

ЗАСІБ СТАТИЧНОГО АНАЛІЗУ .NET ПРОГРАМ ЗА ДОПОМОГОЮ ПЕРЕПИСУВАЛЬНИХ ПРАВИЛ

Т.А. Мамедов, А.Ю. Дорошенко, Р.С. Шевченко

Розроблено програмний засіб, який виявляє в C#-програмах проблеми ресурсоспоживання з погляду роботи з файлами за допомогою переписувальних правил. Для цього використана система TermWare, яку можна легко вбудовувати в ті програмні системи, які побудовані на JVM. Для того, щоб працювати з C#-програмами, був реалізований спеціальний плагін до TermWare, який перекладає програми на мову термів. Цей плагін використовує компілятор Roslyn, який дозволяє знаходити синтаксичні помилки в програмах і дозволяє зосередитися на основній задачі – генерації термів з вихідного коду. Також описаний практичний додаток системи TermWare – статичний аналізатор, який знаходить проблеми відкритих-закритих файлів.

Ключові слова: аналіз ресурсоспоживання, генератор термів, правила переписування, TermWare.

Разработано программное средство, который находит в C#-программах проблемы ресурсопотребления з точки зрения работы с файлами с помощью переписывающих правил. Для этого была использована система TermWare, который можно легко встраивать в программные средства, которые работают на JVM. Для того, чтобы работать с C#-программами был реализованный специальный плагин для TermWare, который переводит программы на язык термов. Этот плагин использует компилятор Roslyn, который позволяет находить синтаксические ошибки в программах и сосредоточиться на основной задаче – генерации термов с исходного кода. Также описано практическое приложение системы TermWare – статический анализатор, который находит проблемы открытых-закрытых файлов.

Ключевые слова: анализ ресурсопотребления, генератор термов, правила переписывания, TermWare.

A software tool that finds problems of resource consumption in case of work with files using rewriting rules was implemented. To reach the goal, the TermWare system, which could be embedded into those systems running on JVM, was used. In order to work with C#-programs, the special plugin for TermWare, which helps to generate appropriate terms from source code, was developed. The plugin uses the Roslyn compiler, which allows users to find syntax errors in programs and focus on the primary task of generating terms from source code. Also, a practical application based on TermWare system – a static analyzer that finds problems with open-close files, was described in the article.

Key words: analysis of resource consumption, terms generator, rewriting rules, TermWare.

Вступ

Багато програмних забезпечень після розробки проходять етап оптимізації та аналізу з погляду швидкодії та ін. Для цього розробники найчастіше використовують власний досвід та ручний спосіб. Але існує багато систем для оптимізації та аналізу програм, які можуть виявити проблемні частини початкового коду. Часто вони вбудовані в інтегроване середовище розробки, але й є різні аналізатори, які працюють як надбудова. Серед таких надбудов є такі: JavaChecker від компанії GradSoft, яка є статичним аналізатором для Java-програм, або Resharper від компанії JetBrains, яка може працювати як інтегроване програмне забезпечення над середовищами Visual Studio та Rider, які призначені для розробки C#-програм.

В роботі описана одна з можливих використань переписувальних правил для аналізу початкового коду програмного забезпечення. Зазвичай, переписувальні правила представляють об’єкти як алгебраїчні терми та використовують трансформацію цих термів за допомогою наборів правил та послідовністю їх виконання. Переписувальні правила є технологією, яка вже підтвердила свою практичність, та використовується для вирішення різноманітних задач [1]. Одним з аргументів щодо актуальності стратегії переписування є існування повноцінних середовищ програмування, які використовують цей метод. Серед таких середовищ програмування можна назвати Maude [2].

В роботі використовувалась система переписувальних правил TermWare [3], яка дозволяє користувачу вбудовувати систему в свою програму, як бібліотеку, та використовувати її для генерації та маніпуляції термами за допомогою операцій над деревовидними структурами програми. Сама система написана як розширення для Java, але вона може використовуватись і для інших мов програмування. Оскільки на мові Java вже існують досить багато систем, з якими програми можуть взаємодіяти, в роботі головна новизна – це використання TermWare на програмах, розроблених на мові C# для аналізу вихідного програмного коду. Таким чином розроблений продукт може бути використаний для великої кількості систем на ринку праці.

Актуальність роботи аргументується наступним чином: по-перше, як було згадано раніше, техніка переписувальних правил є дуже популярною. По-друге, верифікація та аналіз програм з погляду ресурсоспоживання позбавить розробників від потреби додаткової перевірки коду, покращить якість, а також запобіжить майбутнім помилкам в роботі програмного забезпечення. По-третє, проблема відкритих-закритих файлів є новою для C#-програм, тому що навіть один з найбільш використовуваних додатків (Resharper [4]) ігнорує такі помилки у вихідному коді.

Мета роботи – це розробка аналізатора C#-програм, яка вирішує так звану проблему відкритих-закритих файлів за допомогою переписувальних правил.

Задачі на роботу такі:

- дослідити можливість та спосіб використання TermWare для мови C#;
- розробити інтерпретатор, який би TermWare міг використовувати для переведення з C#-програми на мову термів;
- розробити сторонній додаток, який би аналізував мову термів та знаходив би проблему відкритих/ закритих файлів.

Матеріал роботи організований наступним чином: в першому розділі описана система TermWare та принцип роботи з нею; в другому розділі описана реалізація інтерпритатора мови C# на мову термів; в третьому розділі описаний принцип роботи аналізатора та вирішення проблеми відкритих/закритих файлів; останній розділ розкриває майбутні плани та можливі наступні кроки.

Опис системи TermWare

Для аналізу C#-програми використовується система переписувальних правил TermWare. TermWare – це система символьних обчислень для розробки динамічних додатків на основі правил переписування, яка працює з алгеброю термів, яка складається з виразів виду $f(t_1 \dots t_n)$.

TermWare дуже зручна система переписування, оскільки її можна вбудовувати в інші Java програми за допомогою TermWare API.

Основні області застосування даної системи – це:

- побудова проблемно-орієнтованих мов надвисокого рівня;
- системи комп'ютерної алгебри;
- аналіз і перетворення формальних моделей різного роду;
- програмна інженерія [3].

Для завдання перетворень використовуються правила Termware, тобто конструкції виду *source [condition] -> destination [action]*. Тут *source* – вихідний терм (зразок для пошуку), *condition* – умова застосування правила, *destination* – перетворений терм, *action* – додаткове дія при спрацьовуванні правила. Кожен з 4 компонентів правила може містити змінні (які записуються у вигляді $ var), що забезпечує спільність правил. Компоненти condition і action є необов'язковими. Вони можуть виконувати довільний процедурний код, зокрема використовувати додаткові дані про програму [5].

Опис генератора термів для C#-програм

Для роботи з TermWare був розроблений генератор термів для C#-програм, описаний в роботі [5]. Але розробка з тієї роботи не є ідеальною, тому що в ній використовувалися сторонні засоби для роботи з компілятором Roslyn, який розроблений компанією Microsoft, та використовується повсюдно.

Ця проблема була вирішена в цій версії генератора термів. По-перше, всі сторонні засоби як proxygen [6], javonet [7] та інші були замінені на механізм серіалізації та десеріалізації. Якщо говорити більш детально, то проблема використання сторонніх засобів полягала в складності розробки програми з API, які були представлені для користування, і внаслідок – незручний спосіб використання самого генератора термів. А технічна проблема полягала в тому, що Roslyn запускається на CLR, а TermWare – на JVM. Тому результат парсеру, який міг би бути використаним для перевірки правильності синтаксису, а також синтаксичне дерево, раніше були отримані за допомогою сторонніх програм.

Оновлена версія працює за допомогою сериалізації/десериалізації. Але механізм передбачає лише синтаксичний аналіз за допомогою Roslyn та отримання результатів: правильний чи неправильний синтаксис в програмі, яка була подана на вхід. Структурно вихідний файл, який потім десеріалізується в Java виглядає так:

```
{
  "Errors": [
    "(1,23): error CS1733: Expected expression",
    "(1,23): error CS1002: ; expected"
  ]
}
```

Якщо говорити більш детально, то об'єкт компілятора *CompilationUnitSyntax* має важливе поле *Diagnostic*, за допомогою якого можна дізнатись всі синтаксичні проблеми коду.

Після того як за допомогою Roslyn [8] програма виконала синтаксичний аналіз, JSON-файл з результатами десеріалізується в середовищі JVM і далі починається процес побудови абстрактного синтаксичного дерева та генерування термів.

На рисунку показана приблизна та спрощена структура переробленого генератора термів.

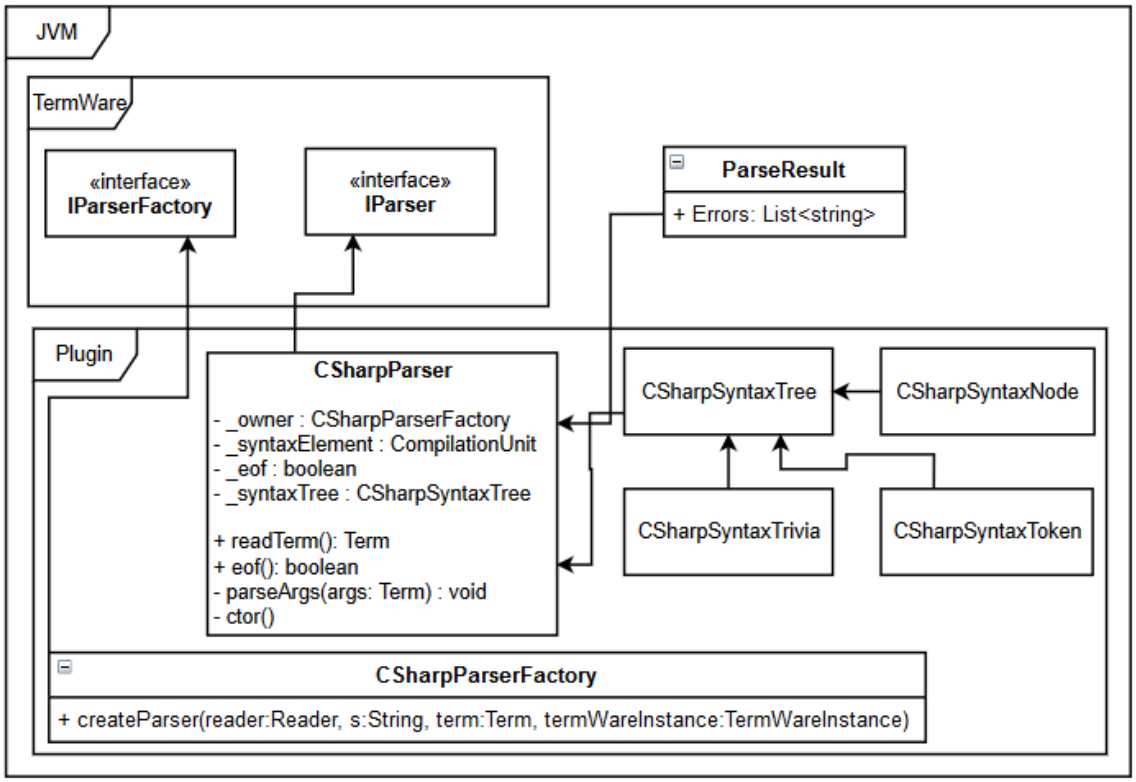

Рисунок. Спрощена діаграма генератора термів

Тепер можна подивитись один з прикладів переведення C#-програми в мову термів. Допустимо, що є наступний клас:

```
using System;

namespace Program {
   public class A
   {
       string Method()
       {
           return “Hello World!”;
       }
   }
}
```

Його інтерпретація у вигляді термального дерева зроблена наступним чином:

```
CompilationUnit(
UsingDeclaration(Name(cons(Identifier(“System”), NIL))),
TypeDeclaration(Modifiers(1, NIL),
   ClassOrInterfaceDeclaration(class, Identifier(“A”), NIL, DerivesList(NIL),
      ClassOrInterfaceBody(cons(
 cons(ClassOrInterfaceBodyDeclaration(
    Modifiers(0),
    MethodDeclaration(NIL,
      String,
      MethodDeclarator(Identifier(“M”), FormalParameters(NIL)), NIL,
Block(cons(ReturnStatement(StringLiteral(“Hello World”)), NIL)))), NIL)
)))))
```

Одним з недоліків, які пов’язані з розробленим генератором термів, але ще не виправлених в цій версії – це підтримка версійності мови. На даний момент, цей генератор термів підтримує C# 1-5 версії. Тому в

майбутніх роботах генератор буде дороблений таким чином, щоб вона підтримувала всі версії мови, включно останню, яка вийшла нещодавно.

Ще одним покращенням, яка буде в наступній версії – це перехід до системи TermWare-3.

Опис аналізатора ресурсоспоживання програм

Одним з можливих способів вирішення проблеми ресурсоспоживання є використання переписувальних правил. TermWare має можливість генерувати не лише термальні абстрактні синтаксичні дерева, а й можливість будувати так звані модельні терми – де до кожного елемента синтаксичного дерева добавлений ще один елемент, який називається контекст. Більш детальний опис модельних термів наведений в роботі [9], який і взятий в основу цієї роботи.

Є багато інших способів аналізувати проблеми в коді, зокрема проблему ресурсоспоживання програм. Наприклад, одним з найвідоміших прикладів аналізу коду є Resharper. Для аналізу слабких сторін вихідного коду Resharper використовує статичний аналіз. Переписувальні правила можуть дати більше, оскільки користувач може зробити додатковий аналіз терму.

Ще одним мінусом Resharper в контексті даної роботи є те, що даний конкурент не має вирішення проблеми відкритих-закритих файлів. Але звісно, що функціонал Resharper поки що переважає результат цієї статті. Це і попередження в місцях, де вихідних код можна оптимізувати з різних боків, і аналіз стилю коду та ін. Але, задачею є більш глибокий аналіз програми з погляду саме ресурсоспоживання.

Іншим способом аналізу C#-програм є перевірка під час виконання програми (runtime inspection). Таку можливість використання показників дає Visual Studio, починаючи з останніх версій. Це можливість подивитись на використання програмою CPU або безпосердньо пам'яті за допомогою Diagnostic Tools [10]. Мінусом цього способу є можливість такого аналізу лише в етапі запуску програми в Visual Studio, також великі витрати з погляду часу для того, щоб розробник спершу аналізував дані, які він отримав за допомогою цього інструментарію, а потім і локалізував місце, де ця проблема виникає. Також одним з мінусів такого підходу є необхідність тестування всіх можливих ситуацій, в яких проблема витоку ресурсів може виникнути. Серед плюсів цього способу в порівнянні з результатом цієї статті є те, що це більш універсальний спосіб знаходження проблеми ресурсоспоживання.

В роботі буде наведена ілюстрація підходу щодо вирішення проблеми витоку ресурсів. Зазвичай, виток ресурсів відбувається, коли програма бере ресурси в операційній системі чи постороннього програмному забезпеченні. В C#-програмах такі класи, які потрібно «закривати» перед тим, як метод закінчить своє виконання, можна ідентифікувати за допомогою інтерфейсів, які вони реалізовують.

Наприклад:

- *IDisposable* – забезпечує механізм звільнення некерованих ресурсів (такі класи, як *FileStream*, *MemoryStream* та ін. реалізовують саме цей інтерфейс);
- *IDataReader* – забезпечує засіб считування одного або безліч потоків, призначених лише для прямої передачі наборів результатів, отриманих виконанням команди у джерелі даних (яскравим прикладом використання цього інтерфейсу є класс SqlDataReader, який використовується для отримання результатів SQL-запиту і після його використання особливо важливим є метод Close()).

Для аналізу таких даних, потрібно для кожної мовної конструкції генерувати множину станів, яку й аналізатор буде обробляти. Для цього є декілька станів обробки:

- Opened*(expression, location)* – визначає місце (*location*) та в C#-програмі змінна або аргумент (*expression*), де потенційно можливий витік ресурсів. При цьому цей стан буде трактуватися, як ще не закритий;
- Cons(*s1, s2*) – позначає те, що один стан виникає після другого стану;
- Return(*s*) – цим символом позначається стан, при якому програмне забезпечення закінчило своє виконання;
- Cond(*v, s*) – означає, що якщо виконується умова *v*, то програма можливо буде в стані *s*;
- *s1* || *s2* – означає, що можливе одне з двух станів, але точно невідомо, яке саме з них;
- *s1* && *s2* – означає, що програма в стані *s1* або *s2* [9].

Програма вважається коректною, тобто аналізатор не знайшов виток ресурсів, якщо фінальний стан не містить Opened().

Наступний код написаний як приклад и виводить зміст текстового файлу на екран консольного додатку:

```
public void OutputFileContents(string path)
  {
        if (string.IsNullOrEmpty(path))
        {
              throw new ArgumentNullException(nameof(path));
        }
```

```
        if (!File.Exists(path))
          {
                Console.WriteLine($"File does not exist in the current path: {path}");
                return;
          }

          try
          {
                var fs = File.OpenRead(path);
                StreamReader reader = new StreamReader(fs);
                string fileContents = reader.ReadToEnd();
                Console.WriteLine(fileContents);
                fs.Dispose();
          }
          catch (Exception ex)
          {
                Console.WriteLine($"The program thrown an exception: {ex.Message}");
                return;
          }
}
```

Наведений приклад є некоректним з погляду ресурсоспоживання, тому що при помилці (викидання «винятку» на етапі читання з файлу в *FileStream* або *StreamReader*), *FileStream* або *StreamReader* не будуть закриті, що може призвести до неочікуваних наслідків. Тобто серед всіх виразів найважливішу роль в аналізі витоків пам'яті грають створення нових об'єктів та визови методів.

При створенні нових об'єктів не можна стовідсотково говорити про проблеми витоку пам'яті навіть якщо створений об'єкт реалізують інтерфейси *IDisposible* або *IDataReader*, оскільки в C# є конструкції такого типу:

```
using(var fs = new FileStream(path))
{
  // expression body
}
```

Такі вирази рівноцінні виразу try, catch, finally, де в останньому блоці викликається Dispose() метод. Саме тому цей вираз можливо використати тільки у тому випадку, коли створюваний клас реалізовує IDisposable.

Тепер розглянемо правила обробки виразу у тих випадках, коли створюється об'єкт на мові C#:

```
CheckExpression($var,
  AllocationExpressionModel(TypeRef($tname,$type),$arguments,$ctx),$state)
  [
        $ctx.subtypeOrSame($type,
            $ctx.resolveFullClassName("System.IDisposable") &&
            !($type.getAttribute("NotDisposable")==true)) ||
        $ctx.subtypeOrSame($type,
            $ctx.resolveFullClassName("System.System.Data.IDataReader")) &&
!($type.getAttribute("NotDisposable")==true)
        ]-> CheckAllocationNotProxy($var,$type,$arguments,$ctx,$state)
        !-> CheckExpressions($arguments,$state),
  CheckAllocationNotProxy($var,$type,$arguments,$ctx,$state)
        -> ((
              !($type.getAttribute("DisposableProxy")==true) ||
              apply(openclose::DisposableArguments, apply(NormalizeExpressions,$arguments))
  ))
  ? cons(OPENED($var,$ctx.getFileAndLine()),$state)
  : CheckExpressions($var,$arguments,$state)
```

Для початку дивимось на умову *subTypeOrSame*, який перевіряє, чи є даний клас, який створюється, успадковує від інтерфейсів IDisposable або IDataReader. Після того, якщо умови виконуються, іде перевірка на *CheckAllocationNotProxy*, а він у свою чергу перевіряє, чи встановлений зовнішній атрибут. Якщо

зовнішній атрибут не встановлений, то він додає стан *OPENED*, в іншому випадку йде перевірка аргументів конструктора.

Висновок

Отже, покращений плагін системи TermWare для перекладання програми, написаної мовою C# на мову термів. У роботі також коротко описана сама система TermWare і принципи її роботи.

Створено аналізатор, який дозволяє виявити проблему відкритих/закритих файлів в реалізованій програмі за допомогою переписувальних правил. Також були наведені інші способи для реалізації такого аналізатора та їх кращі сторони та недоліки.

Також є багато ідей щодо розвитку цього напрямку в контексті мови C# і навіть платформи .NET. Одним з таких ідей є написання транслятора з мови C# на TLA+, який може проверифікувати програму на предмет споживання ресурсів. Іншим планом, який приваблює своєю універсальністю – це аналіз не конкретної мови програмування, а CLR в цілому. Звісно, таким чином всі мови, які об'єднані на платформі .NET, можуть бути проаналізовані і оптимізовані з точки погляду ресурсоспоживання.

Література

1. Dershowitz N., Jouannaud J.-P. (1990). Jan van Leeuwen (ed.). Rewrite Systems. Handbook of Theoretical Computer Science. B. Elsevier. P. 243–320.
2. Winkler T. Programming in OBJ and Maude, in Functional Programming, Concurrency, Simulation and Automated Reasoning, International Lecture Series 1991–1992, McMaster University, Hamilton, Ontario, Canada, by ed. Peter Lauer, Springer Verland. LNCS. P. 229–277.
3. TermWare. URL: http://www.gradsoft.ua/products/termware_rus.html (дата звернення: 18.02.2020)
4. Resharper. URL: https://www.jetbrains.com/resharper/ (дата звернення: 18.02.2020)
5. Мамедов Т.А., Дорошенко А.Ю. Засіб самоналаштування програм на платформі .NET за допомогою переписувальних правил. *Проблеми програмування*. 2019. № 2. С. 11–16.
6. JNI4NET. URL: http://jni4net.com/ (дата звернення: 18.02.2020)
7. Javonet – Java to .NET Bridge, C#, VB.NET. URL: https://www.javonet.com/ (дата звернення: 18.02.2020)
8. Roslyn. URL: https://github.com/dotnet/roslyn (дата звернення: 18.02.2020)
9. Шевченко Р.С., Дорошенко А.Ю. Приминение систем переписывания термов к анализу исходного программного кода. *Проблеми програмування*. 2008. № 2-3. С. 305–312.
10. Quickstart: First look at profiling tools. URL: https://docs.microsoft.com/en-us/dotnet/api/system.data.idatareader?view=netfnetframe-4.8 (дата звернення: 18.02.2020)

References

1. Dershowitz N., Jouannaud J.-P. (1990). Jan van Leeuwen (ed.). Rewrite Systems. Handbook of Theoretical Computer Science. B. Elsevier. P. 243–320.
2. Winkler T. Programming in OBJ and Maude, in Functional Programming, Concurrency, Simulation and Automated Reasoning, International Lecture Series 1991–1992, McMaster University, Hamilton, Ontario, Canada, by ed. Peter Lauer, Springer Verland. LNCS. P. 229–277.
3. Termware. [Online] Available from: http://www.gradsoft.ua/products/termware_rus.html [Accessed: 18 February 2020]
4. Resharper. [Online] Available from: https://www.jetbrains.com/resharper/ [Accessed: 18 February 2020]
5. Mamedov T. Doroshenko A. A method of tuning programs on .NET platform with rewriting rules. Problems of programming. 2019. N 2. P. 11–16. (in Ukrainian).
6. JNI4NET. [Online] Available from: http://jni4net.com/ [Accessed: 18 February 2020]
7. Javonet – Java to .NET Bridge, C#, VB.NET. [Online] Available from: https://www.javonet.com/ [Accessed: 18 February 2020]
8. Roslyn. [Online] Available from: https://github.com/dotnet/roslyn [Accessed: 18 February 2020]
9. Shevchenko R. & Doroshenko A. Using term rewriting systems for source code analysis. *Problems of programming*. 2008. №2-3, P. 305-312. (in Russian).
10. Quickstart: First look at profiling Tools [Online] Available from: https://docs.microsoft.com/en-us/dotnet/api/system.data.idatareader?view=netfnetframe-4.8 [Accessed: 18 February 2020]

Одержано 26.02.2020

***Про авторів*:**

Мамедов Турал Алірзайович,
аспірант Інституту програмних систем НАН України.
Кількість наукових публікацій в українських виданнях – 2.
http://orcid.org/0000-0003-3029-5834,

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук,
професор, завідувач відділу теорії комп'ютерних обчислень
Інституту програмних систем НАН України,
професор кафедри автоматики і управління в технічних системах
НТУУ "КПІ імені Ігоря Сікорського".
Кількість наукових публікацій в українських виданнях – понад 180.
Кількість наукових публікацій в зарубіжних виданнях – понад 60.
Індекс Гірша – 6.
http://orcid.org/0000-0002-8435-1451,

Шевченко Руслан Сергійович,
підприємець.
Кількість наукових публікацій в українських виданнях – 11.
Кількість наукових публікацій в зарубіжних виданнях – 5.
http://orcid.org/0000-0002-1554-2019.

Місце роботи авторів:

Інститут програмних систем НАН України,
проспект Академіка Глушкова, 40,
03187, м. Київ-187,

Національний технічний університет України
"КПІ імені Ігоря Сікорського",

ПП «Руслан Шевченко».

Тел.: (044) 526 3559.
E-mail: tural.mamedov@outlook.com,
doroshenkoanatoliy2@gmail.com,
ruslan@shevchenko.kiev.ua