STUDIA INFORMATICA 2010
Volume 31 Number 3 (91)

Dariusz R. AUGUSTYN, Szymon KUNC
Politechnika Śląska, Instytut Informatyki

MODUŁ TRANSLACJI JĘZYKA MATLAB NA C#, WSPOMAGAJĄCY TWORZENIE PROGRAMÓW SYMULACJI CIĄGŁYCH UKŁADÓW DYNAMICZNYCH, DZIAŁAJĄCYCH W ŚRODOWISKU URUCHOMIENIOWYM .NET

Streszczenie. Artykuł omawia budowę i zastosowanie zaproponowanego translatora języka MATLAB na język C#, w zakresie kodu funkcji opisujących równania stanu ciągłego układu dynamicznego. Translator M2NET, na podstawie kodu M-funkcji, tworzy opis dynamiki w postaci komponentów – pliku źródłowego w języku C# albo biblioteki DLL. Użycie translatora powala na wykorzystanie licznych dostępnych zasobów w postaci skryptów z opisami dynamiki, wcześniej utworzonych i przetestowanych w popularnym środowisku obliczeniowym MATLAB. Rezultaty działania translatora mogą być bezpośrednio wykorzystane przy budowie programów symulacyjnych, działających w środowisku uruchomieniowym .NET Framework. W szczególności mogą być wykorzystane przy tworzeniu efektywnych, zrównoleglonych programów symulacyjnych, zbudowanych na podstawie modułu Parallel Extensions to .NET Framework.

Słowa kluczowe: gramatyka bezkontekstowa, translator, język MATLAB, język C#, model CodeDom, równania stanu, modelowanie ciągłych układów dynamicznych

MATLAB TO C# TRANSLATION MODULE SUPPORTING DEVELOPMENT OF .NET-BASED PROGRAMS FOR SIMULATION OF CONTINUOUS DYNAMICAL SYSTEMS

Summary. The paper describes an architecture and application of a proposed translator from MATLAB to C#. It translates source code of functions implementing state equations of continuous dynamical systems. Using a code of a M-function the translator named M2NET, creates a description of a dynamical system as a C# source code file or a managed library. The translator lets use numerous resources – functions described different dynamical systems, previously created and tested in MATLAB, the popular computing system. Results of the translation can be used directly for creation of .NET-based simulation programs. Particularly, they can be used for

developing effective parallelized simulation programs based on Parallel Extensions to .NET Framework module.

Keywords: context-free grammar, translator, MATLAB language, C# language, CodeDom model, state equations, modeling of continuous dynamical systems

1. Wstęp

Popularność dwóch rozwiązań: systemu MATLAB [2] w zakresie modelowania zachowania układów dynamicznych oraz środowiska .NET jest powodem, dla którego zaproponowano translator kodu języka skryptowego MATLAB na język C#. Użycie translatora pozwala na automatyczne przełożenie opisu dynamiki układu, zadanej w postaci równań stanu.

Dzięki użyciu opisywanego narzędzia, efektywne programy symulacji, działające na platformie uruchomieniowej .NET, mogą wykorzystać opis dynamiki, wcześniej wykonanej w języku MATLAB. W szczególności, możliwe jest skorzystanie, w omawianym zastosowaniu, z wydajnych wielowątkowych programów, wykorzystujących moduł zrównoleglający – Parallel Extensions to .NET Framework [15, 16]. To ostatnie rozwiązanie pozwala na efektywne wykorzystanie mocy obliczeniowej komputerów z procesorami wielordzeniowymi.

Interaktywny system MATLAB może być użyty jako wygodne narzędzie do tworzenia, uruchamiania i testowania. Zasadnicza (czasochłonna) symulacja może odbywać się z użyciem specjalizowanej, efektywnej wersji programu, działającej na platformie .NET.

Ciągły układ dynamiczny, opisany równaniem różniczkowym, może być scharakteryzowany przez układ równań stanu. Stopień równania różniczkowego odpowiada liczbie równań stanu. Równanie stanu to równanie różniczkowe rzędu pierwszego ze względu na jedną zmienną stanu. Lewą stronę równania stanu stanowi pochodna zmiennej stanu.

Funkcja określająca dynamikę układu ciągłego w postaci równań stanu w języku MATLAB [13] ma następującą sygnaturę:

```
function dx = rstanu (t, x),
```

gdzie *rstanu* to przyjęta tutaj przykładowa nazwa M-funkcji [12], t – skalar, chwilowa wartość zmiennej niezależnej, x – wektor wartości zmiennych stanu w chwili t, dx – wynikowy wektor wartości pochodnych zmiennych stanu.

Termin „całkowanie równań stanu” odnosi się do procesu wyznaczania wartości prawych stron równań stanu (czyli wyznaczania wartości pochodnych zmiennych stanu, przez użycie *rstanu*) w ramach działania wybranej funkcji numerycznego całkowania (w MATLABie to funkcje o nazwach: ODE45, ODE23, ODE113, ODE15S, ODE23S, ODE23T, ODE23TB [13], od ang. *ordinary differential equations*). Metody całkowania [1] automatycznie, wielokrotnie wywołują funkcję opisującą równania stanu (tutaj, w podanym przykładzie, *rstanu*),

dostosowując się do zadanych parametrów eksperymentu (zadana tolerancja lub/i krok całkowania). Rezultatem działania są ciągi wartości każdej ze zmiennych stanu w dyskretnych chwilach czasu.

Niniejszy artykuł opisuje budowę i zastosowanie translatora M2NET (MATLAB to .NET), służącego do automatycznej generacji kodu metody opisującej równania stanu. Efekty działania zaproponowanego translatora (pliki źródłowe C# lub zarządzane podzespoły DLL ang. *Dynamic Link Library*) mogą być bezpośrednio użyte do budowy programów symulacji ciągłych układów dynamicznych z użyciem platformy .NET, np. [3]. Otrzymane w ten sposób programy będę na ogół szybsze od swoich MATLABowych odpowiedników.

2. Generatory kompilatorów – wprowadzenie

Generator kompilatorów to narzędzie, które tworzy interpreter, parser lub kompilator na podstawie formalnego opisu gramatyki języka. Najwcześniejszym oraz ciągle najbardziej popularnym w sensie liczby zastosowań typem generatora jest generator parserów, na którego wejście jest podawany opis gramatyki języka programowania, a wynikiem działania jest kod źródłowy parsera realizujący odpowiednią maszynę stanów oraz tablicę przejść między poszczególnymi stanami.

Gramatyki będące przedmiotem zainteresowania w omawianym zastosowaniu to tzw. gramatyki bezkontekstowe. Są one określane przez następujące elementy [4]:

1) zbiór symboli leksykalnych zwanych terminalami (symbolami końcowymi),
2) zbiór symboli nieterminalnych,
3) zbiór produkcji (lista produkcji, lista reguł), z których każda składa się z symbolu nieterminalnego, zwanego lewą stroną produkcji, strzałki oraz sekwencji symboli leksykalnych i nieterminalnych, zwanych prawą stroną produkcji.
4) jeden, wyróżniony symbol nieterminalny, zwany symbolem startowym (głowa gramatyki).

Najczęściej opis gramatyki wejściowego języka opisany jest za pomocą notacji BNF lub EBNF[1].

Na wejście skompilowanego parsera podawane są jednostki leksykalne (zwane również tokenami), które można pogrupować w symbole nieterminalne i poddawać analizie składniowej. Analiza składniowa to proces, który pozwala ustalić, czy dany ciąg symboli leksykalnych może zostać „wygenerowany” przez gramatykę [4]. Wynikiem działania parsera

[1] BNF – Bacus-Naur Form – notacja Bacusa-Naura to sposób zapisu bezkontekstowej gramatyki języka formalnego. EBNF to jedna z rozszerzonych form tej notacji [4].

może być, w zależności od wariantu, powstanie tzw. drzewa parsowania[2] lub konkretne działania (tzw. akcje semantyczne), podejmowane w wyniku interpretacji ciągu symboli leksykalnych. Języki formalne, ich opis i zagadnienia związane z translacją tworzą odrębną, ważną gałąź informatyki.

Obecnie dostępnych jest nawet kilkadziesiąt generatorów kompilatorów [5]. W ramach przeglądu rozwiązań odpowiednich dla omawianego zastosowania brane były pod uwagę popularne, następujące narzędzia programowe: GNU Bison [6] + Flex [7], ANTLR [8], Grammatica [9]. Popularność, obszerna dokumentacja dostępna w Internecie oraz możliwość wygenerowania parsera w języku C# (uwzględniając założenie, że tworzony translator M2NET sam będzie programem dla platformy .NET) przesądziły o wyborze narzędzia Grammatica.

3. Grammatica – charakterystyka zastosowanego narzędzia programowego do generacji parserów

Grammatica generuje parsery do kodu w języku C# lub Java. Narzędzie to jest wolnym oprogramowaniem[3] udostępnianym na licencji GNU LGPL[4].

Poniżej wyszczególniono ważniejsze cechy tego narzędzia:

- Grammatica używa wbudowanego analizatora leksykalnego, a zatem nie jest konieczne stosowanie innych dodatkowych narzędzi w celu wygenerowania parsera (np. tak jak jest to konieczne w przypadku GNU Bison).
- Opis gramatyki umieszczany jest w osobnym pliku z rozszerzeniem .grammar (separacja definicji gramatyki od kodów źródłowych, wynikających z procesu generacji parsera). Standardowo, do opisu składni wykorzystuje się notację EBNF, a do definicji wyrażeń leksykalnych wyrażenia regularne. Opcjonalnie, można użyć do definicji tokenów zwykłych łańcuchów znaków. Dzięki takiemu uproszczeniu znika konieczność stosowania znaków specjalnych (ang. *escape characters*), używanych przy rozpoznawaniu pewnych wyrażeń za pomocą mechanizmu wyrażeń regularnych.
- Grammatica, podobnie jak ANTLR, oferuje wsparcie dla gramatyk LL(k). Obecnie brakuje wsparcia dla gramatyk LALR(1)[5]. Parser LL jest analizatorem składniowym, który

[2] Drzewo parsowania – drzewo wyprowadzenia – wynik przeprowadzenia analizy składniowej zdania [4].

[3] Wolne oprogramowanie (ang. *free software*) [11].

[4] GNU Lesser GPL – licencja bardzo podobna do GNU GPL, z tą różnicą, że umożliwia wykorzystanie i dystrybuowanie danego programu również w projektach, które zamkną udostępniony w ten sposób kod [12].

[5] LALR (ang. *Look Ahead Left-Right*) oznacza wstępującą analizę składniową i jest to jedna z rodzajów analizy LR. W analizie LR parser czyta znaki od lewej do prawej i wyprowadza prawą produkcję. Liczba ujęta w nawiasy (np. LR(k)) oznacza liczbę podglądanych symboli, na podstawie których określana jest każda następna akcja semantyczna analizatora [4].

czyta tekst od lewej do prawej i w przeciwieństwie do parsera LR produkuje lewostronne wyprowadzenie metodą zstępującą. Nie jest dozwolona lewostronna rekurencja na liście produkcji. Liczba tokenów *k*, jaka jest brana pod uwagę w czasie analizy, może się zmieniać i nie jest konieczne deklarowanie jej wartości. W omawianym rozwiązaniu wykorzystano moduł Grammatica w wersji 1.5 [9].

4. Utworzenie parsera w języku C# z użyciem programu Grammatica

W celu wykorzystania opisywanego narzędzia należy wykonać kilka następujących po sobie czynności, które opisano poniżej. Opisywane kroki prowadzą do utworzenia parsera w języku C#, a zatem kroki prowadzące do wygenerowania oraz wykorzystania parsera w języku Java mogą nieznacznie różnić się od tego opisu.

Pierwszym krokiem jest utworzenie pliku z nazwą o rozszerzeniu .grammar (np. *MFunction.grammar*). Plik powinien zawierać opis gramatyki, w tym definicje symboli leksykalnych. Struktura tego pliku wygląda następująco:

```
%header%

GRAMMARTYPE = "LL"
...

%tokens%
...

%productions%
...
```

W sekcji nagłówka (oznaczonej słowem %header%) musi się znaleźć przynajmniej informacja o typie gramatyki. Jak wyżej wspomniano, obecnie Grammatica wspiera tworzenie parserów opartych na gramatykach LL, zatem wartość `GRAMMARTYPE` musi być ustawiona na`"LL"`. Oprócz ustawienia typu gramatyki, w tej sekcji mogą się znaleźć informacje o autorze, wersji, licencji, kilka słów komentarza o samej gramatyce oraz ustawienie, czy parser ma uwzględniać wielkość znaków.

Sekcja tokenów (oznaczona słowem %tokens%) zawiera deklaracje wyrażeń leksykalnych. Wyrażenia takie mogą być opisane jako łańcuchy znaków – deklaracja ujęta w znaki podwójnego cudzysłowu – lub jako wyrażenia regularne – deklaracja ujęta między podwojonymi znakami << oraz >>.
Przykładowa sekcja tokenów:

```
TOKEN_1 = "konkretny lancuch znakow"
TOKEN_2 = <<.>>
```

gdzie TOKEN_2 określony jest jako dowolny znak, z wyjątkiem znaku końca linii.

Kolejny przykład, poniżej, dotyczy fragmentu opracowanego opisu gramatyki języka MATLAB, definiującego następujące tokeny:

```
FUNCTION      = "function"
END           = "end"
IF            = "if"
ELSE          = "else"
ELSEIF        = "elseif"
FOR           = "for"
WHILE         = "while"

ASSIGN        = "="
COLON         = ":"
COMMA         = ","


NAME     = <<[a-zA-Z]+[0-9a-zA-Z]*>>
NUMBER   = <<[0]|[0]\.[0-9]*|[1-9]+[0-9]*(\.[0-9]*)?([eE][0-9]+)?>>
COMMENT  = <<[%].*>>  %ignore%
...
```

Ostatnią sekcją pliku (oznaczoną słowem %production%) jest sekcja definiująca listę produkcji. Lista reguł zdefiniowana jest za pomocą rozszerzonej notacji EBNF. Przykład pokazany poniżej dotyczy fragmentu opracowanego opisu gramatyki języka MATLAB, definiującego listę produkcji:

```
EQUATION = FUNCTION OUTPUT "=" NAME "(" [ PARAMETER ] ")" { STATEMENT }
         ;
OUTPUT = NAME
       | "[" OUTPUT_LIST "]"
       ;
OUTPUT_LIST = NAME { [","] NAME }
            ;
PARAMETER = NAME
          | NAME "," PARAMETER
          ;
STATEMENT = ASSIGNMENT
          | EXPR
          | IF_STATEMENT
          | FOR_LOOP
          | WHILE_LOOP
          | SWITCH_STATEMENT
          | TRY_CATCH_BLOCK
          | BREAK
          | CONTINUE
          | ";"
          ;
ASSIGNMENT = NAME "=" EXPR
           ;
WHILE_LOOP = WHILE EXPR { STATEMENT } END
           ;
TRY_CATCH_BLOCK = TRY { STATEMENT } CATCH NAME { STATEMENT } END
           ;
...
```

Generacja parsera na podstawie zdefiniowanej gramatyki odbywa się przez uruchomienie z linii poleceń programu grammatica-1.5.jar, działającego w środowisku uruchomieniowym Java. Oprócz polecenia służącego do wygenerowania kodu parsera, pomocne mogą być pole-

cenia do weryfikacji poprawności tworzonego rozwiązania. Weryfikację można wykonać pod kątem poprawności samego zapisu gramatyki oraz poprawności logicznej gramatyki. W tym ostatnim przypadku poddaje się próbie parsowania plik zawierający tekst (programu), jaki w zamierzeniu powinien zostać poprawnie sparsowany przez tworzony parser. Poniżej podano przykłady działania programu Grammatica w omawianych trybach tworzenia albo weryfikacji:

- Polecenie sprawdzające poprawność samego opisu gramatyki zawartej w pliku *plik.grammar*:

```
java -jar grammatica-1.5.jar plik.grammar --debug
```

- Polecenie weryfikujące poprawność gramatyki (testowanie „przez przykład”):

```
java -jar grammatica-1.5.jar plik.grammar --parse input_file
```

- Polecenie wykorzystuje opis gramatyki do sparsowania podanego pliku *input_file* (program wypisuje drzewo parsowania do ewentualnego momentu napotkania błędu).
- Polecenie generujące do bieżącego katalogu pliki z kodem parsera w języku C# na podstawie gramatyki opisanej w pliku *plik.grammar*:

```
java -jar grammatica-1.5.jar plik.grammar --csoutput.
```

Ostatnie z wymienionych poleceń, jeśli nie zostaną wykryte błędy, spowoduje utworzenie plików z kodem źródłowym parsera. Powstają cztery pliki o nazwach z prefiksem odpowiadającym nazwie pliku z rozszerzeniem .grammar:

- *Constants.cs* – zawiera wyliczeniowy typ danych, którego pola z przypisanymi wartościami liczbowymi odpowiadają tokenom.
- *Tokenizer.cs* – zawiera definicję klasy *Tokenizer*, która dodaje wzorce rozpoznające zdefiniowane tokeny.
- *Analyzer.cs* – definiuje bazową klasę akcji semantycznych, jakie mają być podejmowane w czasie parsowania; klasa ta wykorzystywana jest w procesie budowy własnego analizatora.
- *Parser.cs* – definiuje klasę parsera; klasa umożliwia utworzenie parsera z domyślnym lub dostosowanym do własnych potrzeb analizatorem; w fazie inicjalizacji parsera tworzone są wzorce poszczególnych produkcji.

Wygenerowane pliki należy dołączyć do tworzonego projektu, zakładając użycie środowiska MS Visual Studio. Aby uzyskać kompilowany moduł parsera, należy dodać w projekcie referencję do biblioteki DLL, dostarczanej razem z opisywanym narzędziem – grammatica-1.5.dll. Biblioteka ta dostarcza między innymi typy bazowe dla wyżej wspomnianych klas parsera, analizatora, tokenizera.

Utworzony w ten sposób parser (w postaci modułu zarządzanego dla platformy .NET) można już wykorzystywać do parsowania tekstów wejściowych. Utworzenie obiektu parsera oraz wywołanie metody *Parse*(), w przypadku braku wykrytych błędów, powoduje zwrócenie korzenia drzewa parsowania.

Uzyskane drzewo parsowania może zostać poddane dalszej analizie. Jednak bardziej efektywną metodą (zamiast przetwarzania drzewa po jego utworzeniu) jest analiza drzewa w trakcie tworzenia, czyli w czasie parsowania. Aby wykonać zamierzone akcje semantyczne, należy utworzyć potomną klasę analizatora i przeciążyć metody wirtualne, które nas interesują. Oczywiście, w ramach akcji semantycznych (implementowanych we własnym analizatorze) będzie tworzony kod dla platformy .NET, adekwatny do MATLABowego odpowiednika. W zależności od przeciążonej metody akcje semantyczne mogą być wykonywane na początku przetwarzania danego węzła lub/i przy jego opuszczeniu. „Włączenie" własnego analizatora odbywa się przez podanie, przy tworzeniu obiektu parsera, obiektu utworzonej, potomnej klasy analizatora.

5. Budowa parsera kodu M-funkcji, definiującej opis dynamiki układu ciągłego

Pliki kodu źródłowego parsera (*MFunctionConstants.sc*, *MFunctionTokenizer.cs*, *MFunctionAnalyzer.cs*, *MFunctionParser.cs*) można uzyskać automatycznie przez wykonanie polecenia:

```
java -jar grammatica-1.5.jar MFunction.grammar --csoutput .
                                               --csnamespace MatlabParser
```

gdzie:

- *MFunction.grammar* to plik opisu gramatyki, nieobiektowego wariantu języka MATLAB w wersji 2007B, obejmującej składnię M-funkcji; fragmenty opisu gramatyki pokazane są w podrozdziale 4; M-funkcja może stanowić opis dynamiki układu ciągłego w postaci równań stanu.
- *MatlabParser* to określenie przestrzeni nazw, do której będą należeć wygenerowane klasy parsera, analizatora i tokenizera.

  Poniżej przedstawiono przykładowy fragment kodu *MFunctionConstants.sc*:

```
internal enum MFunctionConstants {
  FUNCTION = 1001,
  END = 1002,
  IF = 1003,
  ELSE = 1004,
  ELSEIF = 1005,
  ...
  TRY = 1013,
```

```
    CATCH = 1014,
    ...
    WHILE_LOOP = 2026,
     ...
```

Poniżej przedstawiono przykładowy fragment kodu *MFunctionTokenizer.sc*.

```
internal class MFunctionTokenizer : Tokenizer {
  public MFunctionTokenizer(TextReader input : base(input, false) {
    CreatePatterns();
  }
  private void CreatePatterns() {
    TokenPattern pattern;
    pattern = new TokenPattern((int)MFunctionConstants.FUNCTION,
                               "FUNCTION",
                               TokenPattern.PatternType.STRING,
                               "function");
    AddPattern(pattern);
    pattern = new TokenPattern((int)MFunctionConstants.NAME,
                               "NAME",
                               TokenPattern.PatternType.REGEXP,
                               "[a-zA-Z]+[0-9a-zA-Z]*");
    AddPattern(pattern);
    pattern = new TokenPattern((int)MFunctionConstants.COMMENT,
                               "COMMENT",
                               TokenPattern.PatternType.REGEXP,
                               "[%].*");
    pattern.Ignore = true;
    AddPattern(pattern);
    ...
  }
}
```

Cytowany powyżej fragment pliku zawiera przykłady dodania tylko trzech różnych wzorców do definicji analizatora leksykalnego. Wzorzec tokena *FUNCTION* jest łańcuchem znaków, wzorzec *NAME* rozpoznawany jest za pomocą wyrażenia regularnego, a wzorzec *COMMENT* dodatkowo oznaczony jest jako element, który należy ignorować.

Poniżej pokazano sygnatury wygenerowanych metod dla wybranych produkcji domyślnego analizatora, zawarte w pliku *MFunctionAnalyzer.cs*.:

```
internal abstract class MFunctionAnalyzer : Analyzer {
 public override void Enter(Node node);
 public override Node Exit(Node node);
 public virtual void EnterFunction(Token node);
 public virtual Node ExitFunction(Token node);
 ...
 public virtual void EnterName(Token node);
 public virtual Node ExitName(Token node);
 public virtual void EnterNumber(Token node);
 public virtual Node ExitNumber(Token node);
 public virtual void EnterEquation(Production node);
 public virtual Node ExitEquation(Production node);
 ...
```

Przeciążenie wybranych metod domyślnego analizatora pozwala na implementację własnych akcji semantycznych.

Poniżej pokazano kod źródłowy klasy parsera, zawarty w pliku *MFunctionParser.cs*.:

```
internal class MFunctionParser : RecursiveDescentParser {
  public MFunctionParser(TextReader input) : base(input) {
    CreatePatterns();
  }
  public MFunctionParser(TextReader input, MFunctionAnalyzer analyzer)
    : base(input, analyzer){
    CreatePatterns();
  }

  private void CreatePatterns()  {
    ProductionPattern pattern;
    ProductionPatternAlternative alt;

    pattern = new ProductionPattern((int)MFunctionConstants.EQUATION,
                                    "EQUATION");
    alt = new ProductionPatternAlternative();
    alt.AddToken((int)MFunctionConstants.FUNCTION, 1, 1);
    alt.AddProduction((int)MFunctionConstants.OUTPUT, 1, 1);
    alt.AddToken((int)MFunctionConstants.ASSIGN, 1, 1);
    alt.AddToken((int)MFunctionConstants.NAME, 1, 1);
    alt.AddToken((int)MFunctionConstants.LEFT_PAREN, 1, 1);
    alt.AddProduction((int)MFunctionConstants.PARAMETER, 0, 1);
    alt.AddToken((int)MFunctionConstants.RIGHT_PAREN, 1, 1);
    alt.AddProduction((int)MFunctionConstants.STATEMENT, 0, -1);
    pattern.AddAlternative(alt);
    AddPattern(pattern);

    pattern = new ProductionPattern((int) MFunctionConstants.WHILE_LOOP,
                                    "WHILE_LOOP");
    alt = new ProductionPatternAlternative();
    alt.AddToken((int) MFunctionConstants.WHILE, 1, 1);
    alt.AddProduction((int) MFunctionConstants.EXPR, 1, 1);
    alt.AddProduction((int) MFunctionConstants.STATEMENT, 0, -1);
    alt.AddToken((int) MFunctionConstants.END, 1, 1);
    pattern.AddAlternative(alt);
    AddPattern(pattern);
    ...
  }
}
```

Powyższy listing przedstawia kod tworzący wzorce dwóch przykładowych produkcji – *EQUATION* (głowa gramatyki) oraz *WHILE_LOOP*. Obiekt *alt* służy do utworzenia prawych stron reguł.

6. Rozszerzenie analizatora – implementacja akcji semantycznych powodujących generację kodu dla środowiska .NET na podstawie kodu M-funkcji

Kolejnym krokiem prowadzącym do utworzenia translatora M2NET jest zbudowanie (napisanie) własnego analizatora – klasy *CustomAnalyzer*, która rozszerza klasę *MFunctionAnalyzer*. W utworzonej klasie przeciążono 28 metod występujących w *MFunctionAnalyzer*.

W czasie tworzenia drzewa parsowania analizowane są poszczególne jego węzły (przez wywołania odpowiednich metod z *CustomAnalyzer*). W zależności od rodzaju węzła tworzone są odpowiednie konstrukcje językowe przy użyciu danych zawartych w tym węźle oraz wartości, jakie powinny, w przypadku poprawnego programu, zostać przekazane przez jeden lub kilka węzłów „poniżej" aktualnie rozpatrywanego.

W celu przedstawienia elementów programu w danym języku za pomocą pewnej struktury obiektów (tzw. grafu programu), platforma .NET dostarcza pakiet *System.CodeDom* (ang. *Code Document Object Model*) [14]. Przestrzeń nazw *System.CodeDom* dostarcza klas, których funkcjonalności są potrzebne do reprezentacji struktury generowanego programu, niezależnie od składni docelowego języka.

Poniżej pokazano skomentowany fragment kodu utworzonego analizatora, zawartego w pliku *CustomAnalyzer.cs*. Fragment pokazuje tylko kod *ExitEquation* – jednej z przeciążonych metod zdefiniowanych w *MFunctionAnalyzer.cs* oraz kod metody pomocniczej *AddVariableDeclarations*.

```
class CustomAnalyzer : MFunctionAnalyzer  {

    // Słownik zdefiniowanych zmiennych
    private Dictionary<string, ObjectInfo> declaredObjects =
              new Dictionary<string, ObjectInfo>();

    // ExitEquation zwraca głowę gramatyki.
    // Parametr node - węzeł z wygenerowaną metodą
    public override Node ExitEquation(Production node) {
      // Utworzenie metody
      CodeMemberMethod method = new CodeMemberMethod();
      method.ReturnType = new CodeTypeReference(typeof(double[]));
      string methodOutputName = string.Empty;
      // Analiza elementów wchodzacych w sklad metody
      // i dodanie wszystkich instrukcji (statements) ciała metody
      MethodElement methodElement;
      CodeStatementCollection codeStmtCollection =
            new CodeStatementCollection();
      for (int i = 0; i < node.Count; i++) {
         methodElement =
              (MethodElement)StringEnum.Parse(typeof(MethodElement),
            node[i].Name);
        switch (methodElement) {
          case MethodElement.FunctionKeyword:
          case MethodElement.AssignToken:
          case MethodElement.LeftParenthesis:
          case MethodElement.RightParenthesis:
            // Powyższe tokeny bez wpływu na wynik analizy
```

```
            break;
          case MethodElement.Name:
            // Rozponanie nazwy metody
            method.Name = (string)node[i].Values[0];
            break;
          case MethodElement.Output:
            if (node[i].Values.Count > 0){
              methodOutputName = (string)node[i].Values[0];
            }
            break;
          case MethodElement.Parameter:
            // Dodanie parametrów metody
            method.Parameters.AddRange(
                 (CodeParameterDeclarationExpression[])
                 node[i].Values.ToArray(
                       typeof(CodeParameterDeclarationExpression)));
            for (int j = 0; j < method.Parameters.Count; j++){
              if ((declaredObjects[method.Parameters[j].Name].Flag
                       & ObjectFlag.Array) == 0) {
                continue;
              }
              method.Parameters[j].Type =
                 new CodeTypeReference(typeof(double[]));
            }
            break;
          case MethodElement.Statement:
            // Zapamiętanie instrukcji metody
            codeStmtCollection.AddRange(
                 (CodeStatementCollection)node[i].Values[0]);
            break;
        }
      }
      // Ze wzgledu na to, iż Matlab jest dynamicznie typowanym jezykiem,
      // lista zmiennych lokalnych jest tworzona i aktualizowana
      // na bieżaco. Deklaracje rozpoznanych elementów (zmiennych)
      // zostają dodane do ciała metody przed instrukcjami.
      method.Statements.AddRange(AddVariableDeclarations());
      method.Statements.AddRange(codeStmtCollection);

      // Dodanie instrukcji return
      if (methodOutputName != string.Empty
           && declaredObjects.ContainsKey(methodOutputName)){
        CodeMethodReturnStatement returnStmt =
                  new CodeMethodReturnStatement();
        returnStmt.Expression =
                  new CodeVariableReferenceExpression(methodOutputName);
        method.Statements.Add(returnStmt);
      }
      node.Values.Add(method);
      return node;
    } // Koniec ExitEquation

    // Metoda tworząca listę deklaracji zmiennych.
    // Lista deklaracji powstaje na podstawie rozpoznanych zmiennych lokalnych.
    private CodeStatementCollection AddVariableDeclarations() {
          CodeStatementCollection codeStmtCollection
                                = new CodeStatementCollection();
          foreach (var item in declaredObjects) {
              // Jeśli rozpoznana zmienna podczas analizy M-skryptu
              // jest parametrem funkcji, to nie należy jej deklarować
              // w ciele wynikowej metody
              if ((item.Value.Flag & ObjectFlag.Parameter) != 0){
                  continue;
              }
              // Utworzenie nowej deklaracji
              CodeVariableDeclarationStatement declarationExpr
```

```
                    = new CodeVariableDeclarationStatement();
                declarationExpr.Name = item.Key;
                declarationExpr.Type =
                    new CodeTypeReference(item.Value.ObjectType);
                codeStmtCollection.Add(declarationExpr);
            }

            return codeStmtCollection;
        } // Koniec AddVariableDeclarations
    ...
    }
```

Własny analizator tworzy i odpowiednio łączy obiekty klas z przestrzeni nazw *System.CodeDom*, konstruując strukturę grafu programu. Struktura ta, w celu dalszego wykorzystania, jest dodawana do dedykowanego kontenera – *CodeCompileUnit* (patrz rozdział 7).

7. Generacja kodu źródłowego C# na podstawie struktury programu złożonej z obiektów pochodzących z pakietu System.CodeDom

Ostatnim krokiem w tworzeniu translatora jest implementacja czynności generowania wynikowego kodu źródłowego w języku C# (odpowiednika zadanej M-funkcji).

Wynikiem parsowania (z użyciem własnego analizatora) danego tekstu programu w języku MATLAB powinna być jedna metoda .NET. Sposób tworzenia struktur *CodeDom*, reprezentujących tę metodę, został pokazany w poprzednim rozdziale (zobacz treść *ExitEquation*). Metoda ta jest następnie dodawana do definiowanej klasy (domyślnie nazwanej *Equations*) oraz przestrzeni nazw (domyślnie nazwanej *Ode*). Taka zdefiniowana struktura włączana jest do tworzonego obiektu klasy *CodeCompileUnit* (jednostka kompilacji), będącego m.in. kontenerem przestrzeni nazw.

Istnieją dwie opcje wykorzystania tak utworzonego obiektu jednostki kompilacji. W ramach pierwszej opcji informacje z obiektu jednostki kompilacji, za pomocą obiektu klasy *CodeDomProvider*, można przetworzyć bezpośrednio do postaci wykonywalnej, jako zapisanego na dysku pliku DLL lub EXE, albo przechowywanego w pamięci skompilowanego zespołu (ang. *assembly*).

Drugą opcją (podstawową w niniejszym rozwiązaniu) jest generacja kodu źródłowego do pliku tekstowego w wybranym języku platformy .NET. Opisywane w pracy rozwiązanie tworzy na dysku, we wskazanej lokalizacji, plik z kodem w języku C#. Plik ten następnie może zostać zaimportowany do dowolnego projektu (np. w MS Visual Studio), stanowiąc istotny element docelowego programu symulacji ciągłych układów dynamicznych (np. [3]).

Całościowy opis czynności związanych z generacją kodu C# jest następujący: Program translatora tworzy obiekt klasy *CompileUnitGenerator*, który w czasie inicjalizacji tworzy obiekt parsera (z klasy *MFunctionParser*) z odpowiednim (własnym) analizatorem (*CustomAnalyzer*). Konstruktor obiektu *CompileUnitGenerator* automatycznie rozpoczyna parso-

wanie podanego pliku (plik M-funkcyjny) i zapamiętuje referencję do korzenia drzewa parsowania. Wywołanie metody *GenerateCompileUnit* tworzy obiekt jednostki kompilacji (*CodeCompileUnit*), do którego wstawiane są konstrukcje językowe zawarte w drzewie parsowania. Następnie program wykorzystuje instancję klasy *CodeDomProvider* do generacji kodu źródłowego (ewentualnie do kompilacji istniejących źródeł do postaci binarnej) z obiektu jednostki kompilacji. Obiekt ten pozwala na uzyskanie właściwego pliku z tekstem źródłowym programu w języku C#.

8. Przykład zastosowania translatora M2NET

Bardzo prostym przykładem użycia omawianego translatora będzie jego zastosowanie w zadaniu znalezienia rozwiązania następującego równania różniczkowego:

$$x'' - 4x' - 5x = u, \tag{1}$$

gdzie

$$u(t) = \mathbf{1}(t) - \mathbf{1}(t-5). \tag{2}$$

Plik M-funkcyjny (plik *rstanu.m*) w języku MATLAB, opisujący równania stanu (uzyskane metodą ogólną [1]), odpowiadające równaniu różniczkowemu (1) i wymuszeniu (2) ma następującą postać:

```
function dx = rstanu(t, x)
   if t < 5
     u = 1;
   else
     u = 0;
   end
   dx = [ x(2); -4*x(2) - 5*x(1) + u];
```

gdzie zmiennej stanu *x*(1) odpowiada sygnał *x*(*t*), a zmiennej *x*(2) odpowiada *x*'(*t*).

Użycie translatora (w celu weryfikacji poprawności M-pliku) w następujący sposób:

```
M2NET rstanu.m -parse
```

spowoduje wypisanie na standardowe wyjście drzewa parsowania, powstałego w czasie analizy wyżej zadanego pliku (ok. 170 wierszy tekstu).

Użycie translatora (w celu generacji kodu) w następujący sposób:

```
M2NET rstanu.m
```

spowoduje utworzenie pliku *rstanu.cs* z wynikowym kodem w języku C#. Zawartość pliku *rstanu.cs* jest następująca:

```
namespace Ode
{
  using System;
```

```
  public class Equations
  {
    public static double[] rstanu(double t, double[] x)
    {
      double u;
      double[2] dx;
      // Review condition expression below because MATLAB boolean types and C#
ones are not compatible
      if ((t < 5))
      {
        u = 1;
      }
      else
      {
        u = 0;
      }
      dx[0] = x[(2 - 1)];
      dx[1] = ((-4 * x[(2 - 1)]
                (-5 * x[(1 - 1)])
                   + u));
      return dx;
    }
  }
}
```

Translator M2NET został wykorzystany w przykładach modelowania bardziej złożonych układów dynamicznych, np. tych omawianych w [3]. W szczególności może on znaleźć zastosowanie w przypadku symulacji układów opisywanych cząstkowymi równaniami różniczkowymi czy problemu N-ciał (ang. *N-body problem*), czyli układów opisywanych bardzo dużą liczbą równań stanu.

9. Rozwiązanie alternatywne - MATLAB Builder NE

Firma MathWorks Inc. dostarcza rozszerzenie MATLAB Builder NE [17], umożliwiające „konwersję" M-funkcji do modułu .NET. To rozwiązanie stanowi pewną alternatywę dla rozwiązania M2NET. W przeprowadzonej analizie porównawczej użyto modułów MATLAB Version 7.9.0.529 (R2009b) i MATLAB Builder NE Version 3.0.2.

Konwersja z użyciem modułu MATLAB Builder NE polega na stworzeniu podzespołu .NET, który stanowi jedynie opakowanie zasadniczej funkcjonalności, która jest uruchamiana w środowisku runtime-owym MATLABa. Stąd wygenerowane podzespoły DLL są niewielkie w sensie rozmiaru binarium, natomiast istnieje konieczność dystrybucji i instalacji środowiska uruchomieniowego – MATLAB Compiler Runtime (MCR). Rozmiar MCR wynosi 432 MB.

Ponieważ zasadnicze przetwarzanie odbywa się w taki sam sposób jak w środowisku MATLAB, więc efektywność takiego rozwiązania jest zbliżona do raczej niskiej efektywno-

ści wykonania skryptów MATLABa. Praktycznie uzyskano możliwość wywołania funkcji MATLABa z aplikacji .NET, ale potencjalna efektywność platformy .NET nie jest wykorzystana – zasadnicze przetwarzanie nie odbywa się w środowisku .NET.

Producent nie wspomina, czy taki wygenerowany, opakowujący podzespół .NET jest „bezpieczny wielowątkowo" (ang. *MT-safe*), czyli czy mógłby działać poprawnie w warunkach współbieżności, np. takich jak te w [3], gdzie korzysta się z Parallel Extensions to .NET Framework.

Generalnie, narzędzie MATLAB Builder NE ma charakter uniwersalny, ale generowane za jego pomocą rozwiązania nie są ani w całości modułami zarządzanymi (w sensie .NET), ani modułami łatwymi w instalacji i dystrybucji. Stąd dla prostych M-funkcji użycie translatora M2NET może być korzystniejsze.

10. Podsumowanie

Artykuł opisuje budowę, działanie i zastosowanie translatora M2NET (MATLAB to .NET). Na podstawie funkcji, opisującej równania stanu [1] wyrażonej w języku MATLAB, translator umożliwia automatyczną generację odpowiedniego kodu źródłowego metody w języku C#. Umożliwia to wykorzystanie części kodów wcześniej stworzonych w środowisku MATLAB [2], które jest popularnym narzędziem obliczeń naukowo-inżynierskich, m.in. służącym do modelowania ciągłych układów dynamicznych.

Translator pozwala na uzyskanie komponentu opisującego dynamikę układu albo w postaci pliku kodu źródłowego C# (podstawowy tryb działania translatora), albo gotowego podzespołu .NET (zarządzanego modułu DLL). Rezultat generacji może być łatwo wykorzystany w programach symulujących zachowanie ciągłych układów dynamicznych, działających w innym popularnym środowisku uruchomieniowym .NET – zarówno programów sekwencyjnych albo zrównoleglonych [15, 16], np. takich jak te przedstawione w [3], działające efektywnie na maszynach z procesorami wielordzeniowymi.

Praca nad translatorem polegała m.in. na rozpoznaniu i wyborze generatora kompilatora, utworzeniu opisu gramatyki języka MATLAB, dotyczącego budowy kodu M-funkcji oraz zaimplementowaniu własnego analizatora, generującego kod w języku C#. Translator wspomaga tworzenie w środowisku uruchomieniowym .NET programów symulacji układów dynamicznych przez automatyczną generację fragmentów kodu źródłowego. Właśnie te fragmenty najczęściej podlegają zmianie (zmieniają się wraz ze zmianą układu dynamicznego podlegającego badaniom symulacyjnym). Moduł translatora stanowi element uniwersalizujący rozwiązanie omawianie w [3].

BIBLIOGRAFIA

1. Skowronek M.: Modelowanie cyfrowe. Wydawnictwo Politechniki Śląskiej, Gliwice 2008.
2. The MathWorks (2010). MATLAB and Simulink for Technical Computing. http://www.mathworks.com.
3. Augustyn D. R., Kunc S.: Efektywność programów symulacji ciągłych układów dynamicznych, wykorzystujących moduł Parallel Extensions to .NET Framework, uruchamianych na komputerach z procesorami wielordzeniowymi. Studia Informatica Vol. 31, No. 3 (91), Gliwice 2010.
4. Aho A., Ravi Sethi R., Ullman J.: Kompilatory. Reguły, metody i narzędzia. WNT, Warszawa 2002.
5. Comparison of parser generators - Wikipedia (2010). http://en.wikipedia.org/wiki/Comparison_of_parser_generators.
6. Bison - GNU parser generator (2008). http://www.gnu.org/software/bison.
7. flex: The Fast Lexical Analyzer (2010). http://flex.sourceforge.net.
8. Terence Parr. ANTLR Parser Generator (2010). http://www.antlr.org.
9. Grammatica :: Parser Generator (2010). http://grammatica.percederberg.net.
10. Free Software Foundation. The Free Software Definition (2009). http://www.gnu.org-/philosophy/free-sw.html.
11. Free Software Foundation. GNU Lesser General Public License (2007). http://www.gnu-.org/copyleft/lesser.html.
12. Declare function – MATLAB (2010). http://www.mathworks.com/access/helpdesk/help-/techdoc/ref/function.html.
13. Solve initial value problems for ordinary differential equations – MATLAB (2010). http://www.mathworks.com/access/helpdesk/help/techdoc/ref/ode113.html?BB=1.
14. Microsoft. Dynamic Source Code Generation and Compilation (2009). http://msdn-.microsoft.com/en-us/library/650ax5cx.aspx.
15. The Moth - Parallel Extensions (2010). http://www.danielmoth.com/Blog/parallel-extensions.aspx.
16. Microsoft. Parallel Computing - Concurrency, Programming, Processing, Multi-Core (2009). http://msdn.microsoft.com/en-us/concurrency/default.aspx.
17. MATLAB Builder NE (for Microsoft .NET Framework) - Introduction and Key Features (2010). http://www.mathworks.com/products/netbuilder/description1.html.

Recenzent: Dr inż. Maciej J. Bargielski

Wpłynęło do Redakcji 16 czerwca 2010 r.

Abstract

This paper describes a translator form MATLAB to C# – the proposed software tool named M2NET, useful for modeling continuous dynamical systems. It translates MATLAB source code of functions implementing state equations of continuous dynamical systems (a set of state equations is a method of describing system's dynamic which is an alternative to an adequate differential equation [1]).

Using a code of a M-function [12] the translator creates a component implementing a dynamic of a continuous system. It may be either a C# source code file or a managed library (a .NET assembly file).

MATLAB is a well-known and handy system for technical and scientific computing, especially useful for simulating behaviors of dynamical systems. It is very convenient for creating and testing system models. This is a reason of existence numerous resources – proven MATLAB scripts. .NET is a relatively new, very popular and efficient environment for software development [17]. New features like Parallel Extensions to .NET Framework module makes .NET-based programs very efficient, especially when multicore processors are used [15]. Usage of the translator may help to cooperate MATLAB with .NET at building programs for simulating of continuous dynamical systems.

Creating M2NET required inter alia choosing a proper parser generator (Grammatica [9]), defining a grammar of a part of MALTAB language (a grammar of M-function codes), implementing a custom analyzer (for creating of a CodeDom program structure of generated component [14]).

Results of the translation can be used directly for creation.NET-based simulation programs. Particularly, they can be used for developing effective parallelized versions of dynamical systems simulation programs like those ones shown in [3].

Adresy

Dariusz Rafał AUGUSTYN: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, draugustyn@polsl.pl.
Szymon KUNC: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, szymon.kunc@gmail.com.