Реализация на вградения механизъм за извод в междинната форма на SPIDER/CNP програма

Цанко Големанов

Abstract: Control Network Programming (CNP) *is a style of high-level programming that is especially effective for solving problems that have natural graph-like representation of imperative, declarative, or mixed nature. The calculation of the behavior of a SPIDER/CNP program was not carried out by separate module Interpreter. Instead, the logic of interpreter and backtracking functionality are embedded directly in the code of the SPIDER/CNP project. This manipulation is performed by SPIDER-compiler on the step of translating SPIDER program to the intermediate code. The main specifics of Net, State and Arrow compilation and the intermediate code generation are discussed.*

Key words: *Backtracking, Control Network Programming, CNP, SpiderCNP, intermediate code.*

ВЪВЕДЕНИЕ

Фундаменталната част на всяка SPIDER-програма е Управляващата Мрежа (**УМ**). УМ по същество представлява ориентиран граф и е крайно множество подмрежи, една от които е главна. Подмрежите могат да се извикват помежду си, потенциално рекурсивно. Всяка подмрежа се състои от етикетирани възли (състояния), и свързващи ги стрелки. По всяка стрелка може да се постави последователност от "примитиви". Примитивите представляват елементарни действия, еквивалентът на които в традиционните езици за програмиране са потребителски-дефинираните функции. Изпълнението на една SPIDER-програма представлява трасировка на графа-УМ и търсенето на път между началното и финалното състояние. Елементарните действия (примитивите) съдържат код на избран host език (Pascal, C и др) и се изпълняват при придвижването по стрелките, свързващи отделните възли. На практика се прилага вградена стратегия за извод - разширен BACKTRACKING [1], като по-долу ще бъдат разгледани подходите с помощта на които е реализиран този основен, за изпълнението на SPIDER програмата, механизъм [2].

Особености на междинната форма на програмата

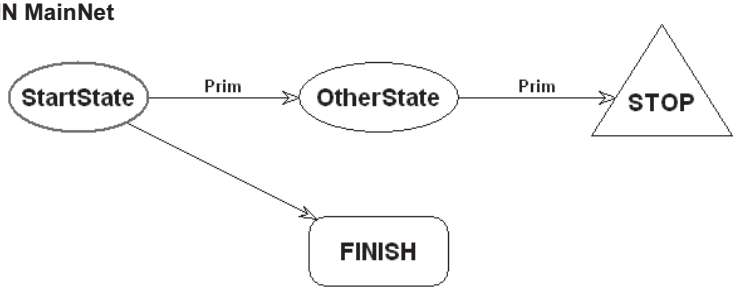
Междинната форма на потребителската програма (SpiderUnit.SPI) се генерира от SPIDER компилатора. Тя представлява програма на host-езика, в която стратегията за извод и механизмите за неговото управление са реализирани с рекурсивни процедури. На всеки от обектите на УМ (подмрежи, състояния, стрелки) съответстват определени програмни структури, а в главната програма се извършва първоначална инициализация и стартиране на изпълнението на УМ.

За да се внесе по-голяма яснота относно вида на междинната програма и реализацията на вградената стратегия за търсене на решение, по-долу е проследена транслацията на малка примерна програма. Без да се влиза в подробности се коментира как изглеждат транслирани подмрежа, състояние и стрелка.

Разглежданата примерна SPIDER програма има една главна подмрежа MainNet, състояща се от две обикновени състояния (StartState, OtherState), две системни състояния (STOP, FINISH) и стрелки, по които е поставен единствения примитив Prim. Кодът на примитива Prim (при host език Pascal), генериращ MessageBox с различни съобщения при движение в права и обратна посока по стрелката, графичното представяне на УМ и съответстващия й текстов вид на са показани на фиг.1.

```
{&P}
 procedure Prim;        // Примитив Prim
 begin
  if FORW then showmessage('Prim forw')
          else showmessage('Prim back');
 end;
```

MAIN MainNet




```
MAIN  MainNet;
body
 StartState:
      Prim        > OtherState
                  > FINISH;
 OtherState:
      Prim        > STOP;
end;
```

Фиг. 1. *Примерна УМ и примитив Prim*

В резултат от работата на SPIDER-компилатора се получава междинна програма на host-езика, първата част на която отговаря изцяло на частта ”Глобални декларации”, а след нея под формата на рекурсивни процедури се разполага транслираната УМ. Тялото на генерираната основна функция SpiderSolutions, чрез която се стартира SPIDER-програмата, има вида показан на фиг.2.

```
function SpiderSolutions : integer;
begin
 __ENT__:='StartState';     //име на входното състояние на главната
                            //подмрежа
 __CPC__:=0;                //инициализиране на текущите разходи на пътя
 __SOL__:=0;                //инициализиране на броя намерени решения
  for __NN__:=1 to 1 do     //за всички дефинирани подмрежи
   begin
    new(__NTO__[__NN__]);
    __NTO__[__NN__]^ := __DO__;        //задаване на мрежови опции по
                                       //подразбиране
   end;
 try
  __N__1(__ENT__,nil,0);               //извикване на главната подмрежа
 except
 end;
 SpiderSolutions := __SOL__;           //връща броя намерени решения
end.
```

Фиг. 2. *Стартираща функция SpiderSolutions*

В главната програма се задават стойности на някои системни променливи, заделя се памет за мрежовите опции и се извършва обръщение към главната подмрежа MainNet (транслирана като процедурата __N__1). От тази процедура се излиза в случай, че главната подмрежа е изпълнена неуспешно.

Транслираната главна подмрежа MainNet е показана на фиг.3.

```
procedure __N__1                          //подмрежа №1 се вика като процедура __N__1
       (__1IS__:string;                   //начално състояние
        __1DR__:pointer;                  //системна информация за изход и обратно връщане
        __1O__:word);
 var __1DA__ : __1DP__;                   //запис за локални данни на подмрежата

begin
 inc(__NTO__[1]^.CurrRec);                //увеличаване на мрежовото ниво на рекурсия
 if __NTO__[1]^.CurrRec-1<=__NTO__[1]^.Recursion then       //проверка за ограничение на
                                                            //нивото
 begin
  new(__1DA__);
  with __1DA__^ do                // установяване на системни и потребителски локални данни
  begin
   Return:=__1O__;
   RetRec:=__1DR__;
   Loops :=nil;
  end;

       if __1IS__='StartState' then __S__1(__1DA__)   //извикване на състояние №1
  else if __1IS__='OtherState' then __S__2(__1DA__)   //извикване на състояние №2
 end;                                                 //неуспешно изпълнение на подмрежата
 dec(__NTO__[1]^.CurrRec);                    //намаляване на мрежовото ниво на рекурсия
end;
```

Фиг. 3. *Транслираната главна подмрежа MainNet*

В тялото на процедурата, отговаряща за една подмрежа се извършва заделяне на памет за системните и потребителски данни на текущото рекурсивно ниво и според стойността на __1IS__ се извършва преход към начално й състояние. Неуспешното изпълнение на това състояние води и до неуспешното изпълнение на подмрежата. Тези действия обаче се извършват само в случай, че текущото ниво на рекурсивна вложеност не надхвърля максимално зададеното чрез системната опция RECURSION [3]. Тогава подмрежата е също неуспешно изпълнена.

Преходът към едно състояние на подмрежа се извършва чрез извикването на съответстващата му процедура. Транслираното състояние StartState е показано на фиг.4.

```
procedure __S__1                          //състояние №1 се вика като процедура __S__1
       (Rec:pointer);                     //указател към запис с мрежовите локални данни
 var
       __SO__1 : __SOP__;                       //локални опции на състоянието
       __SA__1 : __1DP__;                       //достъп до мрежовите локални данни
       __SL__1 : __NCLRP__;                     //локален брояч на примките

begin
 __SA__1:=Rec;                            //предаване на достъп до мрежовите локални данни
 __SO__1:=__FNO__(__NTO__[1],1);          //обновяване на локалните опции на състоянието
 __SL__1:=__FNL__(__SA__1^.Loops,1);            //обновяване на брояча на примките
 inc(__SL__1^.CurrLoops);                             //увеличаване на брояча на примките
 if (__SL__1^.CurrLoops-1<=__SO__1^.Loops) and        //проверка за ограничение на примките
    (__FNV__(__RNV__,1)<=__SO__1^.Visits) then //проверка за еднократно посещение
 begin
  __A__1;                                 //изпълнение на стрелка №1
  __A__2;                                 //изпълнение на стрелка №2 при неуспех на №1
 end;                                     //{неуспешно изпълнение на състояние
 dec(__SL__1^.CurrLoops);                 //{намаляване на брояча на примките
end;
```

Фиг. 4. *Транслирано състояние StartState*

В началото на разгледаната процедура е показано как се използва набор локални системни променливи (за достъп до локалните опции и мрежови данни), след което се извършва последователното изпълнение на излизащите от състоянието две стрелки. Ако първата стрелка (процедура __A__1) се изпълни неуспешно, то се преминава към изпълнение на следващата (__A__2). Неуспешното изпълнение на всички излизащи стрелки води до неуспешното изпълнение на състоянието. Тези действия обаче се извършват само в случай, че текущия брой на примките (повторно влизане в състоянието на едно рекурсивно ниво) не надхвърля максимално зададения (системна опция LOOPS [3]) и не е установена опцията за еднократно посещение (ONEVISIT [3]). Тогава състоянието е също неуспешно изпълнено.

Изпълнението на примитивите по една стрелка се извършва чрез извикване на съответстващата й процедура. Транслираната първа стрелка на състояние StartState е показана на фиг.5.

```
procedure __A__1;                  //стрелка №1 се вика като процедура __A__1
  label __L__1;                    //етикет за начало на обратното изпълнение на стрелката
begin
  failure:=false;  forw:=true;          //инициализация на флаговете: успех, права посока
  __CPC__:=__CPC__+__SO__1^.ArrowCost^;        //увеличаване на текущите пътни разходи
  if __SO__1^.MaxPathCost^>=__CPC__ then       //проверка на максималните пътни разходи
  begin
   PRIM;                                       //право изпълнение на примитива Prim
   if failure then                             //при локален неуспех
    begin
     forw:=false;                         //смяна на посоката на движение по стрелката
     GoTo __L__1                          //преход към обратно изпълнение на стрелката
    end;
   __S__2(Rec);                                //преход към състояние №2 на подмрежата
                                               //неуспешно изпълнение на състояние №2
   forw:=false;                                //смяна на посоката на движение
   __L__1:   PRIM;                             //обратно изпълнение на примитива Prim
  end;                                         //неуспешно изпълнение на стрелка №1
  __CPC__:=__CPC__-__SO__1^.ArrowCost^;        //намаляване на текущите пътни разходи
end;
```

Фиг. 5. *Транслирана първа стрелка на състояние StartState*

Изпълнението на всяка стрелка започва с нулиране на флага за неуспех и установяване на флага за движение в права посока. Следва увеличаване на текущите пътни разходи с тежестта на стрелката и проверка дали получената стойност не надхвърля максимално допустимите разходи, задавани чрез системната опция MAXPATHCOST [3]. В случай че текущите разходи са в рамките на разрешеното, се преминава към последователно изпълнение на примитивите формиращи стрелката, като след всеки се проверява условието за локален неуспех failure. При установяване на такъв, започва обратното изпълнение на преминатите примитиви и това води до неуспешното изпълнение на стрелката. Преходът към друго състояние се състои в извикване на съответстващата му процедура.

Изпълнението на SPIDER УМ приключва с:

- **неуспех** (неуспешно изпълнение на главната подмрежа __N__1) и брой намерени решения в системната променлива __SOL__ = 0;
- **успех** - индикация е достигането до системното състояние FINISH и брой намерени решения в системната променлива __SOL__ > 0.

Програмната обработка на успешното изпълнение (достигнат FINISH) е реализирано като системна процедура със следната алтернативна функционалност:

- Ако е достигнат максималния брой търсени решения:
  - Генериране на изключение (exception), предаване на управлението след try except на стартовата функция SpiderSolutions и изход с

установения брой намерени решения __SOL__
- Иначе продължаване на търсенето на нови решения:
  - Инкрементиране на броя на намерените решения __SOL__
  - Задаване на локален неуспех: **failure** := TRUE
  - Задаване на обратна посока на движение по стрелката: **forw** := FALSE

Трябва да се отбележи, че търсенето на всички възможни решения (при опция SOLUTIONS=ALL [3]) може да предизвика многократни достигания до FINISH, приключващи с неуспешно изпълнение на главната подмрежа __N__1. Обаче и в този случай функцията SpiderSolutions ще изведе коректно намерения текущ брой решения в __SOL__.

ЗАКЛЮЧЕНИЕ

Разгледаната по-горе реализация на основния механизъм за извод в SPIDER/CNP има следните основни предимства:

- Избягва се използването на отделна интерпретираща програма за изпълнение на УМ, понижаваща многократно производителността при изпълнение на мултипарадигмения проект.
- Програмата се състои от два отделни езика - един за описание на примитивите (host) и друг за описание на УМ (SPIDER). Благодарение на компилирането на УМ в базов междинен код на host езика и следващо финално изграждане на проекта, става възможно двата езика могат да се развиват абсолютно независимо.
- Като host език може да се използва някой широко разпространен императивен език с пълните си обектно-ориентирани възможности. Към момента има реализация на SPIDER за Delphi, C++ и Lazarus, като предстои разработка и за Java платформа.

ЛИТЕРАТУРА

[1] R. J. Schalkoff, Intelligent Systems: Principles, Paradigms and Pragmatics. Jones & Bartlett Publishers, 2011.

[2] K. Kratchanov, E.Golemanova, and T.Golemanov, “Control Network Programs and Their Execution,” in 8th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009), Cambridge, UK, 2009, pp. 417–422.

[3] K. Kratchanov, T.Golemanov, and E.Golemanova, “Control Network Programs: Static Search Control with System Options,” in 8th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009), Cambridge, UK, 2009, pp. 423–428.

За контакти:

гл. ас. д-р Цанко Големанов, Катедра “Компютърни системи и технологии”, Русенски университет “Ангел Кънчев”, тел.: 082-888 681, e-mail: TGolemanov@uni-ruse.bg

Докладът е рецензиран.