УДК 681.3.06

СИСТЕМА КОМПОЗИЦИОННО-НОМИНАТИВНОГО ПРОГРАММИРОВАНИЯ SCRIPT.NET

Процик П. П.

Київський національний університет імені Тараса Шевченка,
Київ, проспект Глушкова 2, корп. 6,
E-mail: piter.protsyk@gmail.com

Робота присвячена опису системи прикладного програмування Script.NET. Вона побудована на єдиному методологічному базисі композиційно-номінативного підходу. Її мета – поєднати глибокі математичні основи цього підходу з широкими виразними можливостями сучасних мов програмування, такими як динамічна типізація, метапрограмування, рефлексія, аспектно-орієнтований підхід, механізм контрактів.

Работа посвящена описанию системы прикладного программирования Script.NET. Она построена на едином методологическом базисе композиционно-номинативного подхода. Цель работы – объединить глубокие математические основы данного подхода с широкими выразительными возможностями современных языков программирования, такими как динамическая типизация, метапрограммирование, рефлексия, аспектно-ориентированный подход, механизм контрактов. Исследование их семантики с формальной точки зрения.

The purpose of an article is to introduce a system of applied programming called Script NET. It is constructed on uniform methodological basis of the composition-nominative approach. The purpose of the work is to combine deep mathematical foundations of this approach with wide expressive power of the modern trends in programming languages, such as dynamic typing, metaprogramming, reflection, aspect-oriented approach, mechanism of contracts.

Введение

С момента создания первого вычислительного устройства и до настоящего времени стоит проблема разработки удобных и эффективных средств представления программ, то есть языков программирования. И хотя сущест-вуют тысячи всевозможных языков и диалектов, их количество стремительно растет. Это обусловлено новыми потребностями, накопленным опытом, постоянным усовершенствованием парадигм програм-мирования, открытием новых областей применения вычислительной техники и т.д.

Важным свойством языка является наличие четкой математической семантики. К сожалению не все современные языки удовлетворяют данному требованию. Есть два традиционных решения задачи описания языка: синтактико-семантический и семантико-синтаксический подходы. В первом случае, синтаксис предшествует семантике. Большинство современных языков построено именно таким образом, поэтому формализация их семантики становится весьма нетривиальной задачей, сопряжённой с рядом трудных проблем. Следует отметить, что учитывая это все большее признание завоёвывает семантико-синтаксический подход. При этом сначала формулируется семантика, а затем ей придается необходимая синтаксическая форма. К языкам такого типа, прежде всего, относятся языки функционального программирования: Standard ML, Schema, etc. В силу своей специфики, они так и, не получили такого широкого распространения, как языки построенные на синтактико-семантической основе. Недостатком данных языков, как уже отмечалось, есть то, что их средства построения программ и структурирования данных не достаточно хорошо прописаны и исследованы с математической точки зрения. Поэтому был разработан композиционный [1], а затем и композиционно-номинативный подход [2], являющиеся семантико-синтаксическими по своей сути. Композиционно-номинативный подход основан на трёх основных принципах: развития, композиционности и номинативности. Принцип развития (от абстрактного до конкретного) свидетельствует о последовательном уточнении понятия программы, начиная с наиболее абстрактного определения, продолжая более конкретными и полными уточнениями. Принцип композиционности трактует программы как функции, которые строятся из других функций, называемых базовыми, с помощью специальных операций – композиций. Принцип номинативности устанавливает необходимость использовать отношение именования для работы с программами и данными.

В работе описывается прототип системы программирования Script.NET, построенный на основании композиционно-номинативного подхода.

Среди особенностей системы отметим:

- Семантико-синтаксический подход;

- С-подобный синтаксис базовых композиций;
- Интерпретируемость;
- Номинативные объекты как средства описания данных;
- Средства метапрограммирования с поддержкой рефлексии на уровне языка;
- Поддержка рефлексии;
- Динамическое приведение типов;
- Механизм контрактов;
- Возможность реализации аспекто-ориентированной парадигмы.

Цель работы – на единой методологической основе композиционно-номинативного подхода продемонстрировать современные тенденции проектирования языков высокого уровня, их практическую применимость. Для реализации этого необходимо расширить базовый семантический аппарат похода и придать ему интуитивно понятный синтаксис, чтобы объединить в себе такие черты как простота, интуитивность и при этом высокая выразительная сила.

Далее в работе описывается система Script.NET, даётся введение в мета-программирования, рефлексию, апектно-ориентированный подход, приводятся примеры.

Система Script.NET

Система Script.NET состоит из языка и средств его программной поддержки (реализации). Эти средства свободно распространяются и могут быть загружены с сайта, посвящённого системе [3].

Далее внимание будет сосредоточено на языке системы как элементе, представляющем наибольший интерес. Для этого рассмотрим классический пример программы - реализация алгоритма «пузырьковая сортировка массива элементов»:

```
function sort(a){
  for (i=0; i < a.Length; i=i+1)
   for (j=i+1; j <  a.Length; j=j+1)
    if (a[i] > a[j] )
    {
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
   return a;
 }
a=sort([17, 0, 5, 3,1, 2, 55]);
b=sort(['a','c','z','u','d','g']);
c=sort(['John','Mary','Piter','Victor','Bob']);
```

«Пузырьковая сортировка»

В примере демонстрируется базовый синтаксис, средства создания подпрограмм, возможности динамического приведения типов. Последнее свойство используется для сортирования массивов разнородных элементов с помощью одной функции. В данном примере, во время выполнения программы, при необходимости система автоматически выбирает наиболее подходящий оператор сравнения: для чисел – порядок, заданный на действительной оси, для строк – лексикографический порядок, и т.д.

Рассмотрим более детально отличительные свойства системы.

Номинативные объекты

Это очень гибкий механизм, дополняющий стандартную парадигму объектно-ориентированного подхода. Под номинативным объектом понимается номинативное данное и множество операций языка, определённых на этих данных. Здесь и далее в работе, все неопределённые понятия понимаются в смысле композиционно-номинативного подхода. Все объекты в Script.NET могут рассматриваться именно с такой точки зрения. В языке есть следующие операции:

- Конструктор, **[…], new**

  Служит для создания нового номинативного объекта (НО). Объект может быть создан декларативно или на базе внешнего объекта. Под внешними объектами, в данном случае, понимаются объекты платформы .NET.

  Примеры:
  a = **[** x → 1, y → 2, Text → 'Hello' **]**;
  b = **new** *DataMutant*(form); // будет содержать все поля объекта form
  b.Left = (int)0; // полю Left объекта form будет присвоено значение 0

- Накладка, **CaptureFields, ∇**

  Бинарная операция, которая определена таким образом:
  objectA.CaptureFields(objectB) = {a→ v | a in Names(objectA) / Names(objectB) } Union {a→ v | a in Names(objectB) }

  a = [x → 1, y → 2, Text → 'Hello'];
  b = **new** *DataMutant*(form); // будет содержать все поля объекта form
  b.CaptureFields(a);

- Конвертация, Convert, **:=**

  Операция присваивает полям внешнего объекта, соответствующие поля номинативного объекта. Таким образом, позволяя приводить номинативный объект к любому внешнему типу данных.
  a = [x → 1, y → 2, Text → 'Hello'];
  b **:=** a;

- Доступ к элементу,

  Операция получения доступа к элементу номинативного объекта для записи и изменения:
  a = [x → 1, y → 2, Text → 'Hello'];
  a.z = 15; //создает новый елемент a = $\boldsymbol{a = a \cup [x \to 15]}$
  a.x = 10; //изменяет значение a.CaptureFields([x→10]) или $a = a \nabla [x \to 10]$;
  temp = a.Text; // temp = 'Hello';

Кроме того, специальные операции для работы с внешними объектами: Mutate, Get, Set, Invoke, Resolve и Convert:

Пусть V,U – произвольные внешние объекты, NO1, NO2 – номинативные объекты. Тогда формально данные операции можно определить таким образом:

- NO_1=Mutate(NO, V)= $[fName \to Value \mid V.fName == Value] \nabla\ NO\ \nabla [\text{specName} \to V]$;
- Get(NO, fName) = $\begin{cases} Value, fName \to Value \in NO \\ null, \text{ иначе} \end{cases}$
- V = Resolve(NO) = Get(NO, specName);
- NO_1 = Set(NO, fName,Value) = $NO\ \nabla\ [fName \to Value] \wedge Resolve(NO).fName := Value$;
- U = Convert(NO, V) = Resolve(Mutate(NO, V));

Наиболее близким аналогом номинативных объектов является широкораспространенная техника в современных языках программирования (Python, C#) получившая название DuckTyping. С синтаксической точки зрения, существует подобная нотация JSON. Однако, с номинативными объектами, как с частным случаем более широкого понятия номинативного данного [1] связанна хорошо разработанная математическая теория.

Композиционная структура языка

Композиции – это средства построения программ из базовых элементов: функций, переменных, объектов. В данном случае, композиции – это функции над функциями. Композиционный терм программы задает её семантику. Более подробно описано в [1,2].

В системе существуют композиции таких типов:

- унарные - $Exp \to Exp,\ Stm \to Stm, Exp \to Stm$
- бинарные - $Exp \times Exp \to Exp, Stm \times Stm \to Stm, Exp \times Stm \to Stm,$
- n-арные - $Exp \times Stm \times Stm \to Stm,\ Exp \times Exp \times Exp \times Stm \to Stm,$

где

$$Stm: State \to State,$$

$$Exp: State \to Value.$$

Язык программирования Script.NET содержит расширенный ассортимент композиций:

- **SEQ** или ; – композиция последовательного выполнения инструкций (Sequencing);
- **IF_THEN** – композиция ветвления (Branching);
- **IF_THEN_ELSE** – композиция ветвления с альтернативой;
- **WHILE** – композиция цикла (Looping);
- **FOR** – композиция цикла;
- **FOREACH** – цикл по списку, перечислению;
- **CONV_EXPR** – конвертация выражения в программу;
- **SWITCH** – детерминированный выбор.

Рассмотрим пример программы вычисляющей наибольший общий делитель двух натуральных чисел по алгоритму Евклида:

```
while ( a!=b ){
  if (a>b) a = a-b;
  else if (b>a) b = b-a;
}
```

Построим для этой программы соответствующий композиционный семантический терм:

sem («**while** (a **!=** b) { stms }») = **WHILE** (***sem_e***(«a **!=** b»), ***sem_s***(«stms»)) =
= **WHILE[*NEQ*(state**(a), **state**(b)) , ***sem_s***(«stms»)];
sem_s («**if** (a>b) stm1 **else** stm2») = **IF_THEN_ELSE[*GQ***(state(a), state(b)), ***sems_s***(«stm1»),
sem_s(«stm2»)] = … =

```
IF_THEN_ELSE[
        GQ(state(a), state(b) ),
        ASSIGN[a, DIFF(state(a), state(b) )],
        IF_THEN[
                GQ(state(a), state(b) )
                ASSIGN[b, DIFF(state(b), state(a) )
                ]
        ]
```

sem_s («stm1») = **CONV_EXPR** («a = a - b») = **ASSIGN**(a , ***sem_e***(«a-b»)) =
= **ASSIGN**[a, ***DIFF***(**state**(a), **state**(b))],
sem_s(«stm2») = **IF_THEN**[***GQ***(**state**(a), **state**(b)), **ASSIGN**(b , ***sem_e***(“b-a”))] =
= **IF_THEN**[***GQ***(**state**(a), **state**(b)), **ASSIGN**[b, ***DIFF***(**state**(b), **state**(a))]].

Здесь приняты такие обозначения:

- слова жирными большими латинскими буквами обозначают композиции, **IF_THEN**;
- жирными наклонными большими буквами – функции;
- маленькими жирными буквами – переменные, константы;
- тексты в кавычках « … » - синтаксические конструкции языка;

- ***sem, sem_s, sem_e –*** функции, которые задают семантику программы, инструкции, выражения.

Для вычисления программы необходимо также задать интерпретацию базовых элементов языка **EVAL**.

Так, в реализации, **state –** представляет собой номинативный объект, хранящий текущее состояние памяти.

Далее приводятся примеры интерпретации (вычисления) композиционных термов языка:

$$EVAL(state) = [\, n \to v \mid n \in Names, v \in Values];$$

$$EVAL(\text{ASSIGN}[b, DIFF(\text{state}(b), \text{state}(a))]) = \text{state} \nabla [b \to \text{state}(b) - \text{state}(a)];$$

$$EVAL(GQ(a,b)) = \begin{cases} true, & a \in \mathbb{R}, b \in \mathbb{R}, a > b; \\ false, & a \in \mathbb{R}, b \in \mathbb{R}, a < b; \\ \bot, & \text{в другом случае.} \end{cases}$$

$$EVAL(DIFF(a,b)) = \begin{cases} a - b, & a \in \mathbb{R}, b \in \mathbb{R}; \\ \bot, & \text{в другом случае.} \end{cases}$$

Метапрограммирование и рефлексия языков программирования

Метапрограммирование – это парадигма программирования, при которой программа в результате своей работы генерирует новую программу. Использование метапрограммирования позволяет ускорить процес разработки, улучшить качество получаемого кода програмы. Однако, стоит отметить, что при метапрограммировании необязательно генерируется исходнй код. Например, когда система программирования позволяет программе изменять свой код во время выполнения, как в языках F# [4], Lisp, Smalltalk, JavaScript, Ruby. Если, при этом используются средства исходного языка, говорят, что он обладает возможностью рефлексии.

Рефлексия (или интроспекция) – механизм языка программирования, позволяющий во время выполнения исследовать и изменять структуру, а значит и поведение, программы. Соответствующая парадигма получила название: рефлексивное программирование. Рефлексия характерна для интерпретируемых языков (Python, Ruby, JavaScript, Script.NET), либо выполняющихся на некоторой виртуальной машине (Smalltalk, Java, C#). Рефлексию, можно понимать, как возможность рассматривать выполняемую программу (или часть программы), в качестве данных доступных для обработки (модификации).

На самом низком уровне, машинный код программы, можно также рассматривать рефлексивно, поскольку грань между кодом и данными стирается – все зависит от контекста, в котором ведётся рассмотрение. Обычно инструкции исполняются, а данные обрабатываются, однако программа может трактовать свой код как некоторую область памяти (данное) и рефлексивно вносить изменения. Следует отметить, что такое поведение часто говорит о вредоносности программы, оно характерно для компьютерных вирусов и шпионских программ.

В действительности, по отношению к интерпретатору, либо виртуальной машине высокоуровневая программа тоже является данным. Поэтому, рефлексивность языка при данной необходимости полностью зависит от средств, а не от принципиальных ограничений исполнителя кода.

Приведём классический пример метапрограммы – программа, выводящая свой исходный код. Рассмотрим реализации на разных языках программирования:

- на языке Lisp:

```
((lambda (x) (list x (list 'quote x)))
'(lambda (x) (list x (list 'quote x))))
```

- язык C:

```
int main(void){
char str[]= "int main(void){char str[]= %c%s%c;printf(str, 0x22, str,0x22);}";
printf(str, 0x22, str, 0x22);}
```

- Script.NET:

```
s = ' s="@@"; MessageBox.Show(s.Replace("@@", s)); '; MessageBox.Show(s.Replace('@@', s));
```

Средства метапрограммирования в системе Script.NET имеют свойство рефлексии, то есть для написания метапрограмм используются средства самого языка. Принципиальная возможность такой функциональности обусловлена двумя факторами: методологическим фундаментом и платформой реализации (Microsoft.NET).

Для обеспечения возможностей метапрограммирования в состояние внедряется системный параметр **prog** и значение соответствующее композиционному терму программы:

$$\boldsymbol{state = [\,prog \rightarrow stmts, \ldots\,]}$$

Таким образом, задача написания программы выводящей собственный код в языке Script.NET сводиться к следующему:

```
MessageBox.Show ( prog.Code() );
… код программы …
```

В текущей реализации композиционный терм программы представляется своим размеченным деревом синтаксического разбора. Чтобы его получить, по данной синтаксической конструкции (исходному коду) предусмотрен специальный оператор: ⧏ … ⧐. Например, следующий код:

```
sum = <[ a + b; ]>;
```

даст такой результат: *EVAL*(“sum = <[a + b;]>”) = $\boldsymbol{state \nabla [\,sum \rightarrow SUM(state(a), state(b))]}$.

Далее данный код, можно выполнить в любом месте программы, либо используя текущее состояние, либо используя новое:

```
//выполнить, используя текущее состояние
sum.Execute(state);
//выполнить, используя новое состояние
sum.Execute(new state([a→2, b→3])).
```

Для работы над композиционным термом определены также операции: **INSERT**, **APPEND**, **REMOVE.** Их действие на композиционный терм программы показано на рис. 1,2,3:

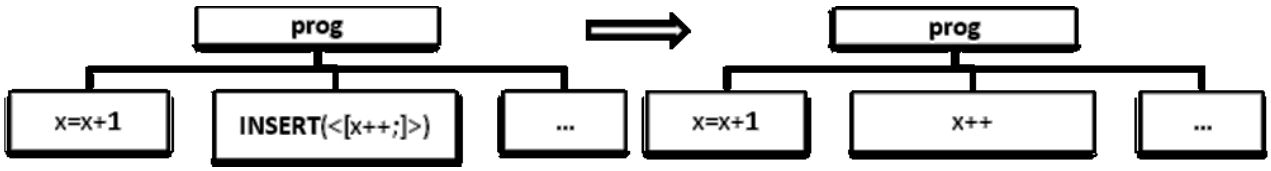

Рис. 1. «Операция INSERT»,

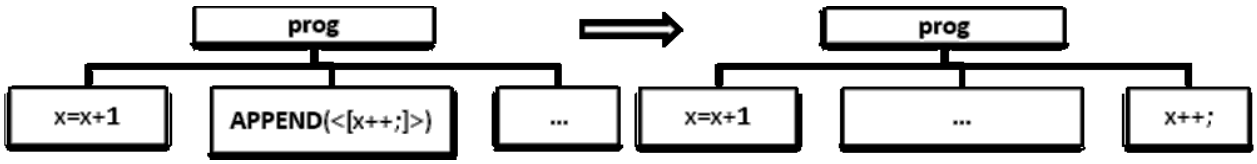

Рис. 2. «Операция APPEND»,

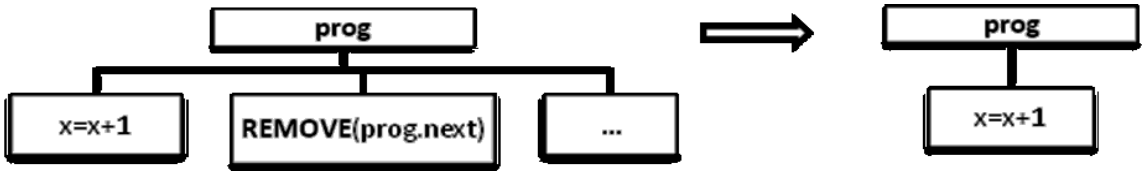

Рис. 3. «Операция REMOVE».

Еще одним элементом рефлексии Script.NET является встроенная функция **eval**(*expression*), которая вычисляет значение выражения заданного строковой константой. Формально функция представляет собой сокращение для выражения - **<[** *expression* **]>**.Execute(**state**).

Аспектно-ориентированное программирование в Script.NET

Аспектно-ориентированное программирование (АОП) [5,6] – это подход к разработке программных систем, который развивает принцип разделения задач (separation of concerns). Разделение задач – это одно из основных правил современного программирования, т.е. каждая отдельная задача должна решаться в отдельном блоке кода (модуле). В структурном программировании, такими блоками являются процедуры и функции. В объектно-ориентированном подходе – класс, который скрывает в себе не только код, но и данные. Однако, некоторые решения не могут быть эффективно реализованы, используя только процедурный или объектно-ориентированный подходы. Например, реализация политики безопасных вызовов методов класса, потребует внесения изменений в каждый метод. Поэтому, политика безопасности это пересекающее отношение (crosscutting concern) – она пересекает весь модуль программной парадигмы, в данном случае – класс. На решение такого рода проблем и направлены усилия аспектно-ориентированного программирования.

Рассмотрим две основных концепции этой парадигмы: аспекты и точки соединения. Аспекты – это пересекающиеся (crosscutting) отношения. Они инкапсулируют особенности поведения взаимно влияющих друг на друга классов в составе повторно используемых модулей. Точка соединения (join point) представляет собой «точно определенную точку в процессе выполнения программы», например, вызов или возврат метода, который может быть обычным возвратом или генерацией исключения. В точки соединения можно внедрять аспекты для изменения поведения целых классов объектов.

Возможность использования аспектной парадигмы целиком зависит от возможностей языка работать с аспектами и точками соединения. У Script.NET есть такие возможности: например, в качестве реализации аспекта можно использовать функцию либо метод какого-либо объекта, в качестве точек соединения – вызов и возврат метода некоторого объекта. Преимущество, которое дает данный подход заключается в том, что исходный код компонентов, к которым применяются аспекты, при этом не подвергается изменениям, а следовательно, можно использовать уже готовые библиотеки классов.

В примере, приведённом в конце раздела, используются средства языка для организации ограниченного по длине стека. Для этого используются номинативные объекты, а также механизм контрактов [7] Script.NET, который позволяет накладывать на функцию пред-, пост- и инвариантные условия с помощью конструкции **[pre** (*conditions*), **post**(*conditions*), **invariant**(*conditions*)**]**. Пред-условия проверяются перед вызовом функции, пост-условия – после, инвариантные – до и после вызова.

Использование номинативных объектов позволяет перехватывать вызов метода внешнего объекта и перенаправить его в другое место в программе. Таким образом, есть возможность отслеживать точки соединения и внедрять аспекты.

Алгоритм работы перехвата вызовов показан графически на рис. 4 и 5.

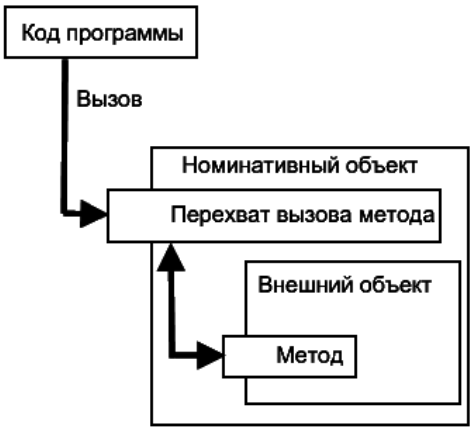

Рис. 4. «Перехват вызова»

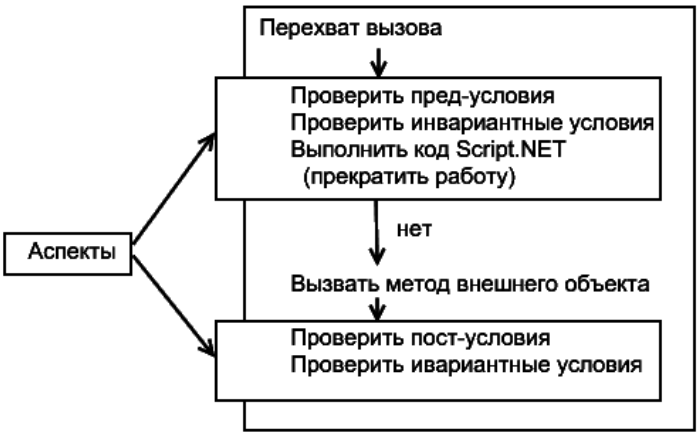

Рис. 5. «Алгоритм перехвата»

Рассмотрим пример использования упомянутой возможности:

```
function PushAspect(item)[
 pre(me.Count < 10 );
 post();  invariant();
]{}

function PopAspect()[
 pre(me.Count > 0);
 post();  invariant();
]{}
// создание стандартного стека
s = new Stack<int>();
// переназначение вызова функции
m = [ Push -> PushAspect, Pop -> PopAspect];
// внедрение объекта
m.Mutate(s);
//вызов функции с внедрённым аспектом из номинативного объекта
m.Push(15);
v = m.Pop();
//вызов приведёт к нарушению предусловия, пользователь будет извещён о
// недопустимости операции
m.Pop();
```

Пример «Механизм контрактов, внедрение аспектов»

Такой механизм перехвата и перенаправления вызова метода объекта можно использовать для реализации кэширования, трассировки, безопасности и получения обновлений.

Примеры использования

Хотя система носит скорее экспериментальный характер, в процессе разработки был обнаружен круг применений, позволяющий оценить преимущества предложенного подхода. Более того, в виду её открытости и доступности через сеть Интернет образовался круг заинтересованных пользователей.

Следует заметить, что языки такого типа, используются как правило, в качестве вспомогательных средств, позволяющих реализовать нестандартную функциональность в пользовательских приложениях. Языковые особенности Script.NET, которые можно условно разделить на такие категории: система типов (динамическая типизация, номинативные объекты), система операций (стандартные композиции, операции над деревом программы, рефлексия кода), декларативная составляющая (пред-, пост-, инвариантные условия) дают возможность более эффективно реализовать то, для чего раньше требовались дополнительные усилия. Среди таких применений языка:

- Использование Script.NET в качестве скриптового языка уровня приложения (на подобии Visual Basic for Applications);
- Использование возможностей номинативных объектов для реализации политик безопасности кода при вызове методов внешних объектов;

- Тестирование визуальных приложений с использованием библиотеки Microsoft UI Automation (.NET Framework 3.0). Динамическая типизация, прозрачный доступ к внешним объектам, изменение их поведения в режиме выполнения позволяют упростить написание тестирующего кода;
- Применение Script.NET в качестве языка генерирования отчетов по шаблонам.

Особое внимание хотелось бы обратить на использование системы в качестве языка программирования поведения агентов в мультиагентных системах. Средства рефлексии и метапрограммирования позволят, в этом случае, агентам легко модифицировать свой код, таким образом, подстраиваясь под изменения среды окружения. Восприятие кода программы, как данного, изначально заложенное в концепцию системы, позволяет организовать прозрачное перемещение агентов в распределённом окружении без затраты дополнительных усилий в противном случае. Возможность декларативного описания условий выполнения операций, позволит повысить надежность работы системы и обеспечить целостность кода.

Сравнение с современными системами программирования

Далее в сводной таблице проводиться сравнительный анализ возможностей Script.NET и других современных языков программирования:

Таблица.

Язык	Расширения ООП	Метапрограм-мирование	АОП	Встроенная верификация	Особенности
Script.NET	✓	✓	✓	✓ / ✗	Семантико-синтаксический подход
C# 3.0	✗	✓ / ✗	✓ / ✗	✗	λ-выражения, LINQ
Python	✓	✓	✓/ ✗	✗	Мульти-парадигменность
Ruby	✓	✓	✓/ ✗	✗	Удачная реализация ООП
F#	✗	✓	✗	✓ / ✗	Функциональность
Spec#	✗	✓ / ✗	✓ /✗	✓	Статическая верификация

Где, ✓ - означает наличие возможности, ✗ - отсутсвие, ✓ / ✗ - частичной поддержки, либо реализации возможности нестандартным путём.

Выводы и дальнейшее развитие системы

Script.NET – это эксперементальная система, основное назначение которой заключается в практическом исследовании современных тенденций в построении языковых интерпретаторов и компиляторов. Разработка новых подходов к решению проблемных задач существующих парадигм программирования.

Дальнейшее развитие системы программирования Script.NET будет вестись сразу в таких направлениях:

- Развитие парадигмы номинативных объектов, расширение функциональных возможностей;
- Усиление возможностей подсистемы метапрограммирования, реализация концепции аспектно-ориентированного программирования в рамках языка;
- Внедрение аппарата формальных методов для статической и динамической верификации программ на подобии того, как это реализовано в системе Spec# [1].

Хотелось бы обратить особое внимание на последний пункт. Данная важная особенность языка будет позволять записывать код вместе с формальными утверждениями относительно свойств, которыми должна удовлетворять программа (формальными спецификациями). В качестве нотации для данной части языка будет использоваться подмножество широко известного метода Z-Notation. Верификация будет проводиться с использованием средств автоматического доказательства теорем (SPASS), и SMT-солверов (Z3, Yaces). Таким образом, в планах на будущее, охватить как можно больше фаз жизненного цикла программ.

1. *Басараб И.А., Никитченко Н.С., Редько В.Н.* Композиционные базы данных. – Киев : ЛИБІДЬ, 1992. – с.192.

2. *Nikitchenko N.S.* Composition Nominative Approach to Program Semantics. – Technical Report IT-TR: 1998-020 [Доклад] / Technical University of Denmark. – 1998. – с. 103.

3. Script.NET. – 2007.11.26, – http://www.codeplex.com/scriptdotnet.

4. *Syme Don.* Leveraging .NET Meta-programming Components from F# [Article] // Proceedings of the 2006 workshop on ML. – Portland : ACM New York. NY, USA, 2006.

5. *Gregor Kiczales*. Aspect-Oriented Programming [Online]. - Xerox Palo Alto Research Center. – 11.26.2007. – www.parc.com/research/projects/aspectj/downloads/ECOOP1997-AOP.pdf.

6. *Полличе Гэри.* Аспектно-ориентированное программирование: Для чего его лучше использовать?. - IBM, 15.03.2006 г. – 25 Ноябрь 2007 г. – http://www.ibm.com/developerworks/ru/library/pollice/index.html.

7. *Meyer Bertrand,* Design by Contract: Making Object-Oriented Programs that Work // Proceedings of the Technology of Object-Oriented Languages and Systems – Tools-25. – IEEE Computer Society, 1997 г.

8. *Barnett Mike*. The Spec# Programming System: An Overview [Conference] // CASSIS. – 2004.