Н. Н. Красильников, krasilnikov@rain.ifmo.ru

Простенький набор классов для описания автоматов

Мне захотелось немного поиграть с перегрузкой операторов и посмотреть, насколько симпатичной можно сделать запись автоматов на C++.

Если у Вас есть идеи, как сделать запись еще более удобной, то я буду рад их услышать :-).

Предложенный набор классов (Приложение 1) позволяет записывать автомат в следующем виде:

```
Auto a;
State s0, s1, s2, s3;

a << s0 << s1 << s2 << s3;

s0 >> s1 | 0 | x1 + x2  | z1      ;
s1 >> s2 | 1 | x1 * !x2 | z1      ;
s2 >> s3 | 0 | x1       | z1 , z2 ;
s3 >> s0 | 0 | x2       | z2      ;
```

`Auto a` – автомат, который в последствии можно будет вызывать по событию **`e`** как **`a(e)`,** где **`e`** – целочисленный номер события.

`State s0, s1, s2, s3` – состояния, которые затем объявляются как состояния автомата **`a`**: **`a << s0 << s1 << s2 << s3`**. Первое добавленное состояние автоматически становится начальным.

`s0 >> s1 | 0 | x1 + x2 | z1` – задание перехода из состояния **`s0`** в состояние **`s1`** по событию «ноль» при условии, что входная переменная **`x1`** или **`x2`** истинна. На переходе выполняется выходная переменная **`z1`**.

Отдельно стоит сказать про переменные **`xi`** и **`zi`** - они являются обернутыми в классы **`FX`** и **`ZX`** указателями на функции вида **`bool fxi()`** и **`void fzi()`**.

Эти классы создавались в основном **just for fun :)**. Также было интересно изучить возможности *C++* в области создания приятных глазу записи различных конструкций, в свете развития языков программирования предметной области.

По наглядности представленная запись уступает графу переходов, но данный набор классов может быть полезен в случае автоматического создания автоматов во время работы приложения или при необходимости верификации при инициализации автомата.

В приложении 2 приведен пример использования предложенных классов.

Приложение 1. Классы для реализации автоматов

Auto.h

```
//
//   Набор классов для описания автоматов
//
//   FX x1(fx1); // Обертка для входной переменной
//   FX x2(fx2);
//
//   FZ z1(fz1); // Обертка для выходной переменной
//   FZ z2(fz2);
//
//   Auto a; // Автомат
//   State s0, s1, s2, s3; // Состояния
//
//   a << s0 << s1 << s2 << s3;
//   // s0 - начальное состояние автомата “a”
//
//   // Описание переходов автомата
//   s0 >> s1 | 0 | x1 + x2  | z1      ;
//   s1 >> s2 | 1 | x1 * !x2 | z1      ;
//   s2 >> s3 | 0 | x1       | z1 , z2 ;
//   s3 >> s0 | 0 | x2       | z2      ;
//
//   Вызов автомата с событием e (типа int):
//   a(e);
//

#include "List"

// Тип входных переменных
typedef bool (*Xi)();

// Тип выходных переменных
typedef void (*Zi)();

// Оберточный класс для входных переменных
class FX
{
public:
     FX(Xi x);
     FX(FX *x);
     FX(const FX &x);
     FX& operator=(const FX &x);
     ~FX();
     FX operator + (FX &f);
     FX operator * (FX &f);
     FX operator ! ();
     bool operator () ();

private:
```

```
     friend class Branch;

     enum{AND,OR};

     bool leaf;
     Xi xs;
     int operation;
     FX *leftOperand;
     FX *rightOperand;
     bool not;

     FX();
};

// Оберточный класс для выходных переменных
class FZ
{
public:
     FZ(Zi z);
     FZ operator + (FZ f);
     FZ operator , (FZ f);
     void operator () ();

private:
     friend class Branch;

     std::list<Zi> zs;

     FZ();
};


class State;

// Класс перехода из одного состояния в другое
class Branch
{
public:
     Branch& operator | (FX x);
     Branch& operator | (FZ z);
     Branch& operator | (int ev);

private:
     friend class State;
     friend class Auto;

     int e;
     State *s;
     State *t;
     FX xs;
     FZ zs;

     Branch ();
```

```
     Branch (State &begin, State &end, int evt, FX x, FZ z);
};

class Auto;

// Класс состояния
class State
{
public:
     State();
     ~State();

     void operator += (Branch &b);
     Branch& operator >> (State &s);

private:
     friend class Auto;

     int number;
     Auto* a;
     std::list<Branch *> branches;
};

// Класс автомата
class Auto
{
public:
     Auto();

     void operator += (State &s);
     Auto& operator << (State &s);

     void Event(int e);
     void operator () (int e);

private:
     int state;
     int maxState;
     std::list<State *> states;
};
```

Auto.cpp

```
#include "Auto.h"

// Оберточный класс для входных переменных

FX::FX()
{
     leftOperand = 0;
     rightOperand = 0;
     not = false;
}

FX::FX(Xi x)
{
     leaf = true;
     xs = x;

     leftOperand = 0;
     rightOperand = 0;
     not = false;
}

FX::FX(FX *x)
{
     leaf = x->leaf;

     if(leaf)
     {
          xs = x->xs;

          leftOperand = 0;
          rightOperand = 0;
     }
     else
     {
          operation = x->operation;

          leftOperand = new FX();
          rightOperand = new FX();

          *leftOperand = *(x->leftOperand);
          *rightOperand = *(x->rightOperand);
     }
}

FX::FX(const FX &x)
{
     leaf = x.leaf;

     if(leaf)
     {
```

```
          xs = x.xs;

          leftOperand = 0;
          rightOperand = 0;
     }
     else
     {
          operation = x.operation;

          leftOperand = new FX();
          rightOperand = new FX();

          *leftOperand = *(x.leftOperand);
          *rightOperand = *(x.rightOperand);
     }
}


FX& FX::operator=(const FX &x)
{
     leaf = x.leaf;
     not = x.not;

     if(leaf)
     {
          xs = x.xs;

          leftOperand = 0;
          rightOperand = 0;
     }
     else
     {
          operation = x.operation;

          leftOperand = new FX();
          rightOperand = new FX();

          *leftOperand = *(x.leftOperand);
          *rightOperand = *(x.rightOperand);
     }

     return *this;
}

FX::~FX()
{
     if(leftOperand != 0) delete leftOperand;
     if(rightOperand != 0) delete rightOperand;
}

FX FX::operator + (FX &f)
{
     FX newf;
```

```
     newf.leaf = false;

     newf.leftOperand = new FX();
     *newf.leftOperand = *this;
     newf.rightOperand = new FX();
     *newf.rightOperand = f;
     newf.operation = OR;

     return newf;
}

FX FX::operator * (FX &f)
{
     FX newf;

     newf.leaf = false;

     newf.leftOperand = new FX();
     *newf.leftOperand = *this;
     newf.rightOperand = new FX();
     *newf.rightOperand = f;
     newf.operation = AND;

     return newf;
}

FX FX::operator ! ()
{
     not = !not;

     return *this;
}

bool FX::operator () ()
{
     if(leaf)
          if(not)
               return !xs();
          else
               return xs();
     else
          if(operation == OR)
               return (*leftOperand)() || (*rightOperand)();
          else
               return (*leftOperand)() && (*rightOperand)();
}


// Оберточный класс для выходных переменных

FZ::FZ()
{
```

```
}

FZ::FZ(Zi z)
{
     zs.push_back(z);
}

FZ FZ::operator + (FZ f)
{
     FZ newf;

     newf.zs = zs;
     newf.zs.merge(f.zs);

     return newf;
}

FZ FZ::operator , (FZ f)
{
     FZ newf;

     newf.zs = zs;
     newf.zs.merge(f.zs);

     return newf;
}

void FZ::operator () ()
{
     for(std::list<Zi>::iterator i = zs.begin();
            i != zs.end(); i++)
     {
          (*i)();
     }
}


// Класс перехода из одного состояния в другое

Branch::Branch ()
{
}

Branch::Branch (State &begin, State &end, int evt, FX x, FZ z)
{
     s = &begin;
     t = &end;

     e = evt;

     xs = x;
     zs = z;
}
```

```
Branch& Branch::operator | (FX x)
{
     xs = x;
     return *this;
}

Branch& Branch::operator | (FZ z)
{
     zs = z;
     return *this;
}

Branch& Branch::operator | (int ev)
{
     e = ev;
     return *this;
}


// Класс состояния

State::State()
{
     number = -1;
     a = 0;
}

State::~State()
{
     for(std::list<Branch *>::iterator i = branches.begin();
            i != branches.end(); i++)
     {
          delete *i;
     }
}

void State::operator += (Branch &b)
{
     branches.push_back(&b);
}

Branch& State::operator >> (State &s)
{
     Branch* b = new Branch();

     b->s = this;
     b->t = &s;

     branches.push_back(b);

     return *b;
}
```

```
// Класс автомата

Auto::Auto()
{
     maxState = -1;
     state = 0;
}

void Auto::operator += (State &s)
{
     if(s.number == -1)
     {
          maxState++;
          s.number = maxState;
          s.a = this;
     }

     states.push_back(&s);
}

Auto& Auto::operator << (State &s)
{
     *this += s;

     return *this;
}

void Auto::Event(int e)
{
     for(std::list<State *>::iterator i = states.begin();
               i != states.end(); i++)
     {
          if((*i)->number == state)
          {
               for(std::list<Branch *>::iterator j =
                  (*i)->branches.begin();
                   j != (*i)->branches.end(); j++)
               {
                    if((*j)->e == e)
                    {
                         if((*j)->xs())
                         {
                              (*j)->zs();
                              state = (*j)->t->number;

                              return;
                         }
                    }
               }
          }
     }
```

```
}

void Auto::operator () (int e)
{
     Event(e);
}
```

Приложение 2. Пример использования

Main.cpp

```
#include "Auto.h"

// Пример использования

bool fx1() {return true;}
bool fx2() {return false;}

void fz1() {}
void fz2() {}

int main()
{
     FX x1(fx1);
     FX x2(fx2);

     FZ z1(fz1);
     FZ z2(fz2);

     Auto a;
     State s0, s1, s2, s3;

     a << s0 << s1 << s2 << s3;

     s0 >> s1 | 0 | x1 + x2  | z1      ;
     s1 >> s2 | 1 | x1 * !x2 | z1      ;
     s2 >> s3 | 0 | x1       | z1 , z2 ;
     s3 >> s0 | 0 | x2       | z2      ;

     a(0);
     a(0);
     a(1);
     a(0);

     return 0;
}
```