Санкт-Петербургский государственный институт точной механики и оптики
(технический университет)

Кафедра «Компьютерные технологии»

М.И. Гуисов, А.Б. Кузнецов, А.А. Шалыто

Задача Д. Майхилла «Синхронизация цепи стрелков»

Объектно-ориентированное программирование
с явным выделением состояний

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2003

Содержание

1. Постановка задачи

В настоящей работе демонстрируется эффективность применения объектно-ориентированного программирования с явным выделением состояний, в котором совместно используются объектно-ориентированный и автоматный стили программирования [1]. В работе [2] был предложен шаблон, который позволяет реализовывать системы автоматов, вызываемых последовательно. Однако, при решении задач с параллельными процессами этот шаблон неприменим, в основном, из-за того, что в нем отсутствует переобозначение переменной состояния.

Для устранения указанного недостатка в настоящей работе используются следующие подходы [3]:

- переобозначение переменной состояния;
- разбиение цикла работы каждого автомата на три этапа – выбор перехода на автоматном графе, переход с выполнением действий, обновление (переобозначение) переменной состояния. Каждый из этапов выполняется одновременно для всех автоматов.

При создании программы для упрощения взаимодействия параллельно работающих автоматов применялся механизм обмена сообщениями, реализованный с помощью библиотеки «swmem» - SWitchMesssageExchangeMechanism [3]. Эта библиотека содержит классы «Сообщение» (CSMessage), «Очередь сообщений» (CSQueue), «Автомат» (CSAutomate), «Массив состояний автомата» (CSStateArray) и структуру «Состояние» (CSAutoState).

Класс «Автомат» является базовым для всех автоматных классов. В нем, в частности, реализуется третий этап цикла работы каждого автомата - обновление переменной, в то время как первый и второй этапы реализуются в классах-наследниках. Кроме того, в наследниках используется предложенный в работе [3] подход к распознаванию сообщений от различных источников – процедура трансляции сообщений (Translate).

При проектировании программы использовалась нотация, предложенная в работе [2].

В качестве примера выбрана задача о синхронизации цепи стрелков, предложенная Д.Майхиллом в 1957 г. [4].

Сформулируем эту задачу. Предположим, что стрелкам, выстроенным в цепь, разрешена следующая процедура общения между собой.

1. Каждый стрелок может общаться непосредственно только со своими ближайшими соседями слева и справа. При этом левофланговый также может общаться с офицером.

2. Сеансы общения могут происходить по одному разу в каждую единицу времени. При этом считается, что офицер и каждый из стрелков имеют часы, которые синхронизированы между собой. Синхронизации часов недостаточно для синхронности выстрела, так как число стрелков в цепи априори не известно.

3. Сеанс общения стрелка состоит в том, что он передает своим соседям по одному условному знаку и принимает к сведению те условные знаки, которые ему сообщают соседи.

В некоторый момент левофланговый получает от офицера приказ **ОГОНЬ**, который адресован всем стрелкам и должен быть выполнен ими одновременно. Это обеспечивается передачей сигнала по цепи до ее конца и возвратом этого сигнала к левофланговому стрелку.

В зависимости от того могут ли стрелки «считать до N» (номер стрелка) и какие знаки могут передаваться в цепи, задача имеет разные решения, отличающиеся по сложности.

Если счет запрещен и в качестве условных знаков нельзя передавать номер стрелка в цепи, то может быть применен алгоритм В.И.Левенштейна. Его модификация описана в работе [5]. В этом алгоритме каждый стрелок

реализуется автоматом с восемью состояниями. Основная идея этого алгоритма состоит в «делении отрезка цепи пополам». По цепи распространяется множество сигналов с задержками 2^i-1, где i - номер сигнала. При встрече некоторого сигнала с первым происходит деление цепи. От места встречи распространяются новые сигналы с теми же номерами. Автоматы, находящиеся в месте встречи, переходят в состояние готовности, предшествующее «синхронному» состоянию. Идея Левенштейна позволила распространять любое число сигналов по цепи автоматов с конечным, и как отмечено выше, небольшим числом состояний.

В настоящей работе, как и в работах [6,7] разрешен «счет до N», а общение стрелков допускает передачу номера стрелка.

Каждый стрелок в цепи должен руководствоваться следующими правилами.

1. Левофланговый стрелок получив приказ, запоминает число один (свой порядковый номер) и через единицу времени сообщает его соседу справа.

2. Если сосед слева сообщил стрелку число **K-1**, то стрелок запоминает свой порядковый номер (**K**) и через единицу времени передает его соседу справа, или, если стрелок - правофланговый, то он отвечает соседу слева о своей готовности и приступает к обратному счету (**K**, **K-1**, ...).

3. Если порядковый номер стрелка равен **K** и сосед справа сообщил ему о своей готовности, то через минуту стрелок передает сигнал о готовности налево и приступает к обратному счету (**K**, **K-1**, ...). Левофланговый сообщает о своей готовности офицеру.

4. Досчитав до нуля, стрелок открывает огонь, нажимая курок один раз за единицу времени.

В результате применения этих правил первый синхронный выстрел (залп) произойдет через **2N** единиц времени после получения левофланговым приказа офицера.

2. Диаграмма основных классов

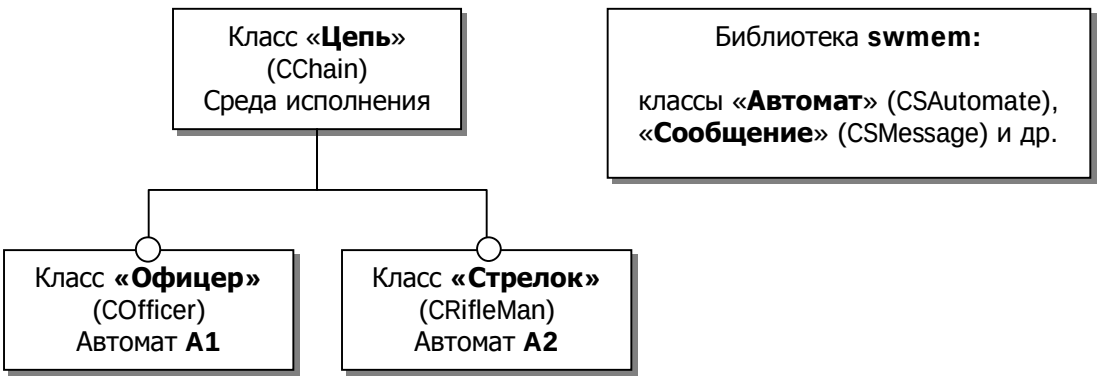

Рис. 1. Диаграмма основных классов

Обратим внимание, что в программе используются N экземпляров класса «Стрелок» и по одному экземпляру остальных классов. Классы «Офицер» и «Стрелок», реализующие автоматы A1 и A2, наследуют автоматный шаблон «Автомат».

Кроме указанных на диаграмме классов, в программе применяются и другие классы, например, «Очередь сообщений», «Состояние» (необходимо для протоколирования) и т.д.

3. Класс «Цепь»

3.1. Словесное описание

Этот класс является главным в приведенной выше иерархии классов. Основной его задачей является связь действий офицера и солдат в шеренге. Он отвечает также за протоколирование работы программы, вызывая соответствующие процедуры классов «Офицер» и «Стрелок».

3.2. Структурная схема класса

Рис. 2. Структурная схема класса «Цепь»

4. Класс «Офицер»

4.1. Словесное описание

Этот класс реализует офицера, отдающего приказы солдату, стоящему на левом фланге. Кроме того, класс (как наследник библиотечного класса «Автомат») протоколирует состояния автомата **A1**, управляющего действиями офицера.

4.2. Структурная схема класса

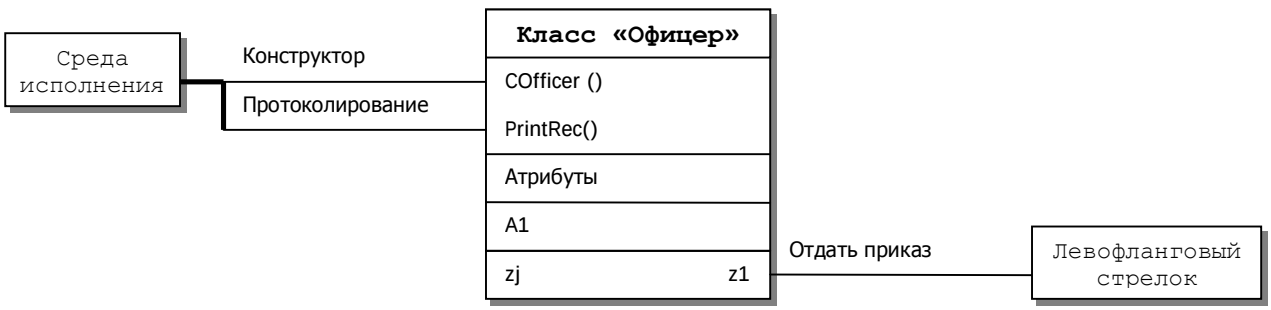

Рис. 3. Структурная схема класса

4.3. Автомат управления действиями офицера

4.3.1. Словесное описание

В начале работы программы офицер находится в **ИСХОДНОМ** (DREAM) состоянии. При получении сообщения **e1** от среды офицер отдает команду левофланговому стрелку. При этом офицер переходит в состояние **ПРИКАЗ** (ORDER). В этом состоянии он ждет готовности левофлангового стрелка (**e2**). Готовность означает, что вся цепь уже готова к стрельбе. После этого офицер переходит в состояние **ОБЗОР** (OBSRV). Переход в **ИСХОДНОЕ** состояние происходит после доклада левофлангового солдата об окончании стрельбы (**e3**).

4.3.2. Схема связей

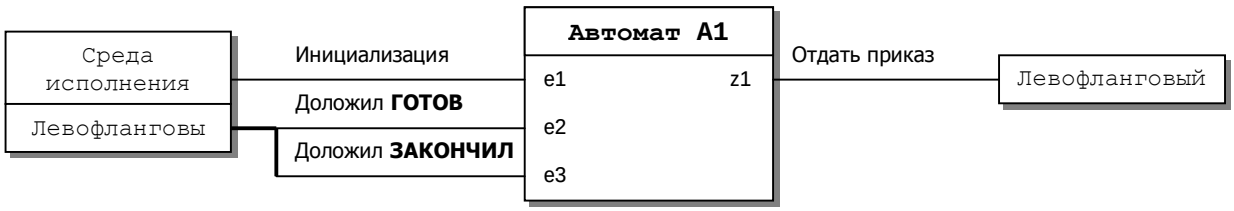

Рис. 4. Схема связей автомата A1

4.3.3. Граф переходов

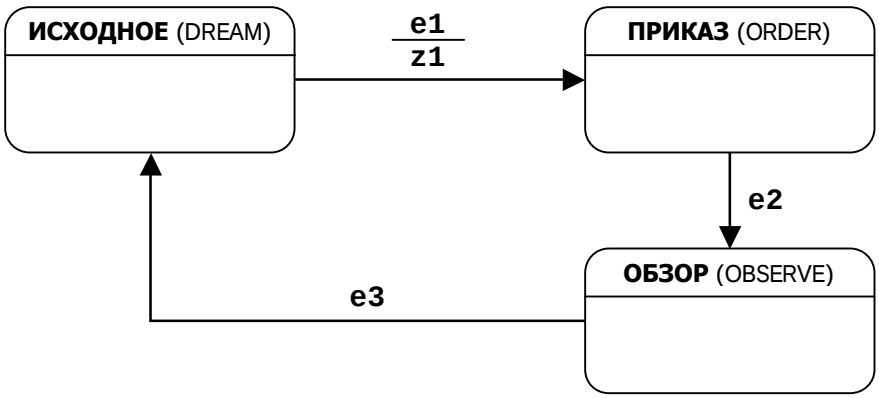

Рис. 5. Граф переходов автомата A1

5. Класс «Стрелок»

5.1. Словесное описание

Этот класс реализует стрелка, выполняющего приказ. Он обеспечивает связь стрелка с ближайшими соседями. Кроме того, класс (как наследник библиотечного класса «Автомат») протоколирует состояния автомата **A2**, управляющего действиями стрелка.

5.2. Структурная схема класса

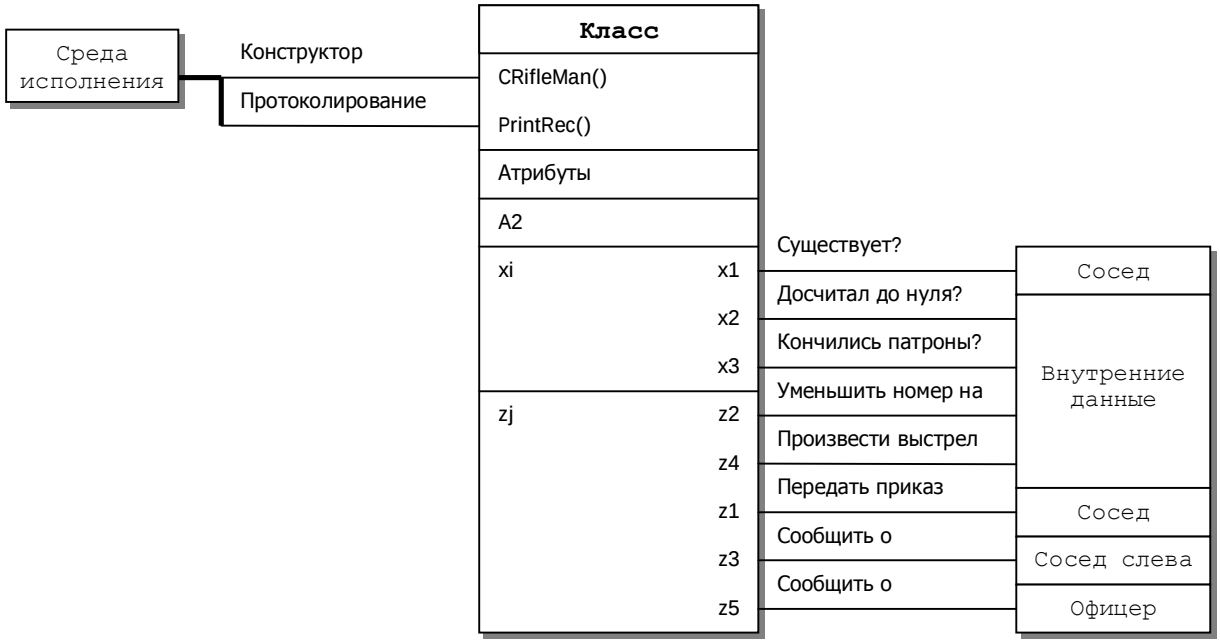

Рис. 6. Структурная схема класса «Стрелок»

5.3. Автомат управления действиями стрелка

5.3.1. Словесное описание

Стрелок, как и офицер, в начале находится в **ИСХОДНОМ** состоянии. Получив приказ от соседа (**e1**) (левофланговый получает приказ от офицера), стрелок запоминает свой порядковый номер, передает приказ направо и переходит в состояние **ПРИКАЗ.** В этом состоянии стрелок находится до тех пор, пока сосед справа не сообщит ему о своей готовности (**e2**). После этого стрелок переходит в состояние **ГОТОВ** (правофланговый переходит в состояние **ГОТОВ** без ожидания). В состоянии **ГОТОВ** стрелок для обеспечения синхронизации считает от своего порядкового номера до нуля. Досчитав, стрелок переходит в состояние **ОГОНЬ**, в котором он находится пока не кончатся патроны в магазине, делая один выстрел в единицу времени. После этого стрелок сообщает о завершении стрельбы и переходит в **ИСХОДНОЕ** состояние.

5.3.2. Схема связей

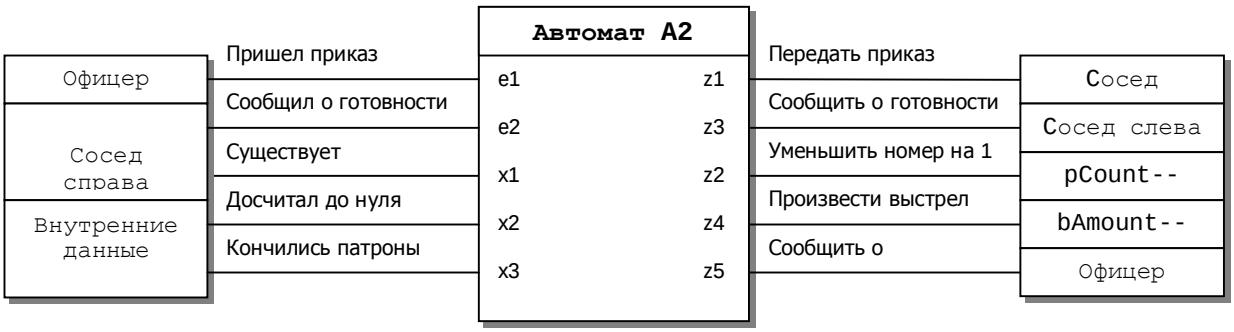

Рис. 7. Схема связей автомата A2

5.3.3. Граф переходов

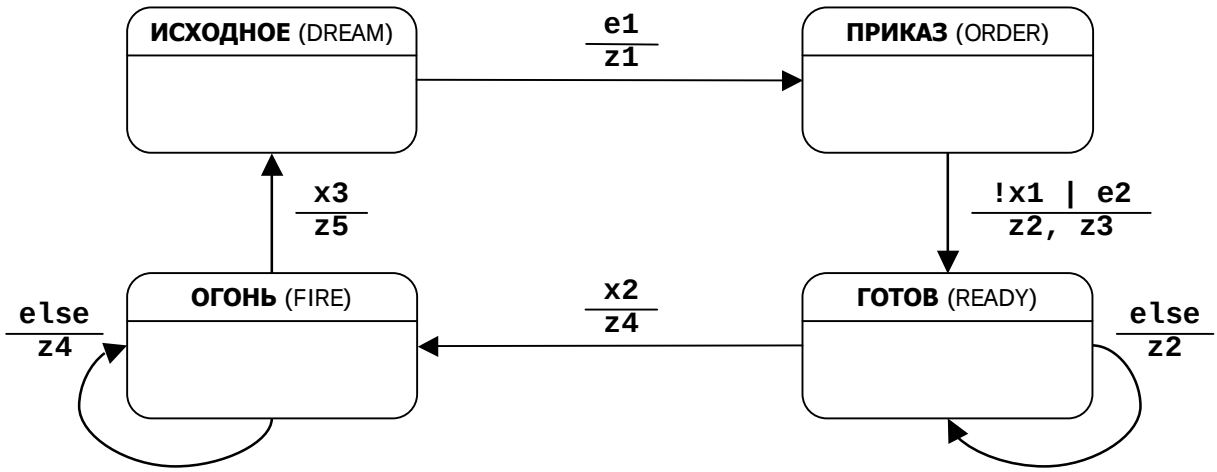

Рис. 8. Граф переходов автомата A2

6. Протокол работы программы

(4 стрелка, 3 патрона в обойме)

```
00. logic entrance
{ A1   :  0 'DREAM' ->  0 'DREAM', с действием -1 }
{ A2_01:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_02:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_03:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_04:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3

01. logic entrance
{ A1   :  0 'DREAM' ->  1 'ORDER', с действием  1 }
{ A2_01:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_02:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_03:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_04:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3

02. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  0 'DREAM' ->  1 'ORDER', с действием  1 } bullets 3/3
{ A2_02:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_03:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_04:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3

03. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_02:  0 'DREAM' ->  1 'ORDER', с действием  1 } bullets 3/3
{ A2_03:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_04:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3

04. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_02:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_03:  0 'DREAM' ->  1 'ORDER', с действием  1 } bullets 3/3
{ A2_04:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3

05. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_02:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_03:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_04:  0 'DREAM' ->  1 'ORDER', с действием  1 } bullets 3/3

06. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_02:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_03:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_04:  1 'ORDER' ->  2 'READY', с действием  3 } bullets 3/3

07. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_02:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_03:  1 'ORDER' ->  2 'READY', с действием  3 } bullets 3/3
{ A2_04:  2 'READY' ->  2 'READY', с действием  2 } bullets 3/3

08. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  1 'ORDER' ->  1 'ORDER', с действием -1 } bullets 3/3
{ A2_02:  1 'ORDER' ->  2 'READY', с действием  3 } bullets 3/3
{ A2_03:  2 'READY' ->  2 'READY', с действием  2 } bullets 3/3
{ A2_04:  2 'READY' ->  2 'READY', с действием  2 } bullets 3/3
```

```
09. logic entrance
{ A1   :  1 'ORDER' ->  1 'ORDER', с действием -1 }
{ A2_01:  1 'ORDER' ->  2 'READY', с действием  3 } bullets 3/3
{ A2_02:  2 'READY' ->  2 'READY', с действием  2 } bullets 3/3
{ A2_03:  2 'READY' ->  2 'READY', с действием  2 } bullets 3/3
{ A2_04:  2 'READY' ->  2 'READY', с действием  2 } bullets 3/3

10. logic entrance
{ A1   :  1 'ORDER' ->  2 'OBSRV', с действием -1 }
{ A2_01:  2 'READY' ->  3 'FIRE ', с действием  4 } bullets 2/3
{ A2_02:  2 'READY' ->  3 'FIRE ', с действием  4 } bullets 2/3
{ A2_03:  2 'READY' ->  3 'FIRE ', с действием  4 } bullets 2/3
{ A2_04:  2 'READY' ->  3 'FIRE ', с действием  4 } bullets 2/3

11. logic entrance
{ A1   :  2 'OBSRV' ->  2 'OBSRV', с действием -1 }
{ A2_01:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 1/3
{ A2_02:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 1/3
{ A2_03:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 1/3
{ A2_04:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 1/3

12. logic entrance
{ A1   :  2 'OBSRV' ->  2 'OBSRV', с действием -1 }
{ A2_01:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 0/3
{ A2_02:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 0/3
{ A2_03:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 0/3
{ A2_04:  3 'FIRE ' ->  3 'FIRE ', с действием  4 } bullets 0/3

13. logic entrance
{ A1   :  2 'OBSRV' ->  2 'OBSRV', с действием -1 }
{ A2_01:  3 'FIRE ' ->  0 'DREAM', с действием  5 } bullets 0/3
{ A2_02:  3 'FIRE ' ->  0 'DREAM', с действием  5 } bullets 0/3
{ A2_03:  3 'FIRE ' ->  0 'DREAM', с действием  5 } bullets 0/3
{ A2_04:  3 'FIRE ' ->  0 'DREAM', с действием  5 } bullets 0/3

14. logic entrance
{ A1   :  2 'OBSRV' ->  0 'DREAM', с действием -1 }
{ A2_01:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_02:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_03:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
{ A2_04:  0 'DREAM' ->  0 'DREAM', с действием -1 } bullets 0/3
```

7. Заключение

Рассмотренный в работе пример продемонстрировал эффективность применения предложенного в работе [3] подхода при реализации систем с параллельными процессами. Использование протоколов позволяет визуализировать работу программы, что значительно облегчает ее отладку и демонстрирует в наглядном виде правильность ее работы.

Список литературы

1. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. N2.

2. *Шалыто А.А., Туккель Н.И.* SWITCH-технология - автоматный подход к созданию программного обеспечения "реактивных" систем //Программирование. 2001. N5. http://is.ifmo.ru/.

3. *Гуисов М.И., Кузнецов А.Б., Шалыто А.А.* Интеграция механизма обмена сообщениями в Switch-технологию. СПбГИТМО (ТУ), 2003, http://is.ifmo.ru, раздел «Проекты»

4. *Goto E.* A Minimum Time Solution of the Firing Squad Problem //Dittoed Course Notes for Applied Mathematics. Harvard Univ., 1962.

5. *Варшавский В.И., Мараховский В.Б., Песчанский В.А., Розенблюм Л.Я.* Однородные структуры. М.: Энергия, 1973.

6. *Трахтенброт Б.А.* Алгоритмы и вычислительные автоматы. М.: Советское радио, 1974.

7. *Любченко В.С.* «Батарея, огонь!» или задача Майхилла для Microsoft Visual C++. О синхронизации процессов в среде Windows // Мир ПК. 2000. N2.

Приложения. Листинги программы

Файл «cFire.cpp»

```
#include "cChain.h"

int main(int argc, char* argv[])
{
	FILE *FD= fopen("log.txt", "w+");	// Указатель на файл протокола
	CChain *fChain = new CChain(FD);

	for (int i=0; i<20; i++) {
		if (i == 1) fChain->DoCommand();
		fChain->LogicSW();
		fChain->Output(i);
	}
	fclose(FD);
	delete fChain;
	return 0;
}
```

Файл «cChain.h»

```
#ifndef _CCHAIN_H_INCLUDED
#define _CCHAIN_H_INCLUDED

#include "cOfficer.h"
#include "cRifleMan.h"
#include "swmem.h"

class CChain {
public:
	CChain(FILE *FD);			// Конструктор
	~CChain();				// Деструктор

	void DoCommand();			// Приказ "сверху"
	void LogicSW();				// Логика класса
	void Output(int step);			// Протоколирование

private:
	CSQueue *Queue;				// Очередь сообщений
	COfficer *Officer;			// Офицер
	CRifleMan *(*rmArray);			// Массив стрелков
	FILE *FDat;
};

#endif
```

Файл «cChain.cpp»

```
#include <stdio.h>
#include "cChain.h"

CChain::CChain(FILE *FD)
{
	this->FDat = FD;
	Queue = new CSQueue();
	rmArray = new CRifleMan* [rmNumber];
	char rmName[10];

	for (int i=0; i<rmNumber; i++) {
		sprintf(rmName, "A2_%.2d", i+1);
		rmArray[i] = new CRifleMan(Queue, FD, i+1, rmName);
	}
```

```
	Officer = new COfficer(Queue, FD);
}

CChain::~CChain()
{
	for (int i=0; i<rmNumber; i++) delete rmArray[i];
	delete [] rmArray;
	delete Officer;
	delete Queue;
}

// Приказ "сверху"
void CChain::DoCommand()
{
	Queue->AddMessage(new CSMessage(1, -1));
}

// Логика класса
void CChain::LogicSW()
{
	CSMessage *msg;
	Officer->Update();
	for (int i=0; i<rmNumber; i++) rmArray[i]->Update();
	Queue->AddMessage(new CSMessage());
	while (!Queue ->IsEmpty()) {
		msg = Queue->GetMessage();
		Officer->A(*msg);
		for (i=0; i<rmNumber; i++) rmArray[i]->A(*msg);
		delete msg;
	}
	Officer->S();
	for (i=0; i<rmNumber; i++) rmArray[i]->S();
}

// Протоколирование
void CChain::Output(int step)
{
	fprintf(FDat, "\n\n%.2i. logic entrance", step);
	Officer->PrintRec();
	for (int i=0; i<rmNumber; i++) {
		rmArray[i]->PrintRec();
	}
}
```

Файл «cOfficer.h»

```
#ifndef _COFFICER_H_INCLUDED
#define _COFFICER_H_INCLUDED

#include "swmem.h"

class COfficer: public CSAutomate {
public:
	COfficer(CSQueue *queue,  // Указатель на очередь сообщений
		FILE *f_rec);	  // Указатель на ОТКРЫТЫЙ файл для записи протокола
	void A(CSMessage &msg);   // Процедура выбора перехода в автомате A1
	void S();                 // Процедура, осуществляющая переход в автомате A1

private:
	int Translate(CSMessage &msg);   // Транслятор сообщения
	void z1();                       // Отдать приказ
};


#endif
```

Файл «cOfficer.cpp»

```
#include <stdio.h>
#include "cOfficer.h"

static const int officer_snum = 3;
static int   officer_narr[officer_snum] = {0, 1, 2};
static char* officer_carr[officer_snum] = {"DREAM", "ORDER", "OBSRV"};


COfficer::COfficer(CSQueue *queue, FILE *f_rec)
:CSAutomate(queue, "A1   ",
            new CSStateArray(officer_snum,  officer_narr, officer_carr), f_rec, true)
{
}


// Процедура выбора перехода в автомате A1
void COfficer::A(CSMessage &msg)
{
      int ev = Translate(msg);
      switch (y_o) {
      case 0:
            if (ev == 1) { y_n = 1; action = 1; }
            break;

      case 1:
            if (ev == 2) { y_n = 2; }
            break;

      case 2:
            if (ev == 3) { y_n = 0; }
      }
}

// Процедура, осуществляющая переход с выполнением действия в автомате A1
void COfficer::S()
{
      switch (action) {
      case 1: z1();
      }
}

// Транслятор сообщения
int COfficer:: Translate(CSMessage &msg)
{
      if (msg.id == 1 && msg.src == -1) return 1;
      if (msg.id == 2 && msg.src ==  1) return 2;
      if (msg.id == 3 && msg.src ==  1) return 3;
      return 0;
}

void COfficer::z1()
{
      Queue->AddMessage(new CSMessage(1, 0));
}
```

Файл «cRifleMan.h»

```
#ifndef _CRIFLEMAN_H_INCLUDED
#define _CRIFLEMAN_H_INCLUDED

#define mSize      3                  // Размер магазина
#define rmNumber   4                  // Количество стрелков

#include "swmem.h"
```

```
class CRifleMan: public CSAutomate {
public:
	int pID;                    // Личный номер стрелка
	int pNumber;                // Переменная отсчета

	CRifleMan(CSQueue *queue,   // Указатель на очередь сообщений
		FILE *f_rec,        // Указатель на ОТКРЫТЫЙ файл для записи протокола
		int rmID, char *rmName);
	void A(CSMessage &msg);     // Процедура выбора перехода в автомате A2
	void S();                   // Процедура, осуществляющая переход в автомате A2
	void PrintRec();

private:
	int bAmount;                // Количество оставшихся в магазине пуль
	int Translate(CSMessage &msg);   // Транслятор сообщения
	bool x1();                  // Есть сосед-стрелок справа?
	bool x2();                  // Отсчет закончен?
	bool x3();                  // Кончились патроны?

	void z1();                  // Зарядить винтовку, передать приказ (e1)
	void z2();                  // Уменьшить номер на 1
	void z3();                  // Сообщить о готовности (e2)
	void z4();                  // Произвести выстрел
	void z5();                  // Сообщить о завершении стрельбы (e3)
};

#endif
```

Файл «cRifleMan.cpp»

```
#include <stdio.h>
#include "cRifleMan.h"

static const int rifleman_snum = 4;
static int    rifleman_narr[rifleman_snum] = {0, 1, 2, 3};
static char* rifleman_carr[rifleman_snum] = {"DREAM", "ORDER", "READY", "FIRE "};

CRifleMan::CRifleMan(CSQueue * queue, FILE *f_rec, int rmID, char *rmName)
:CSAutomate(queue, rmName, new CSStateArray(rifleman_snum, rifleman_narr,
	rifleman_carr), f_rec, true)
{
	pID = rmID;
	bAmount = 0;
}

// Процедура выбора перехода в автомате A2
void CRifleMan::A(CSMessage &msg)
{
	int ev = Translate(msg);
	switch (y_o) {
	case 0:
		if (ev == 1)                    { y_n = 1; action = 1; }
		break;

	case 1:
		if (x1() || ev == 2)            { y_n = 2; action = 3; }
		break;

	case 2:
		if (x2())                       { y_n = 3; action = 4; }
		else                            { action = 2; }
		break;

	case 3:
		if (x3())                       { y_n = 0; action = 5; }
		else                            { action = 4; }
	}
}
```

```
// Процедура, осуществляющая переход с выполнением действия в автомате A2
void CRifleMan::S()
{
	switch (action) {
	case 1: z1(); break;
	case 2: z2(); break;
	case 3: z2(); z3(); break;
	case 4: z4(); break;
	case 5: z5(); break;
	}
}

// Протоколирование
void CRifleMan::PrintRec()
{
	CSAutomate::PrintRec();
	fprintf(f_rec, " bullets %i/%i", bAmount, mSize);
}

// Транслятор сообщения
int CRifleMan:: Translate(CSMessage &msg)
{
	if (msg.id == 1 && msg.src == pID-1) return 1; // e1: Сосед слева передал приказ
	if (msg.id == 2 && msg.src == pID+1) return 2; // e2: Сосед справа сообщил о
	return 0;                                      // готовности
}

bool CRifleMan::x1()
{
	return (pID == rmNumber);
}

bool CRifleMan::x2()
{
	return (pNumber == 0);
}

bool CRifleMan::x3()
{
	return (bAmount == 0);
}

void CRifleMan::z1()
{
	pNumber = pID;
	bAmount = mSize;
	Queue->AddMessage(new CSMessage(1, pID));     // e1: Передаем приказ
}

void CRifleMan::z2()
{
	pNumber--;
}

void CRifleMan::z3()
{
	Queue->AddMessage(new CSMessage(2, pID));     // e2: Сообщаем о готовности
}

void CRifleMan::z4()
{
	bAmount--;
}

void CRifleMan::z5()
{
	Queue->AddMessage(new CSMessage(3, pID));     // e3: Сообщаем об окончании огня
}
```

Файл «swmem.h»

```
#ifndef _SWITCH_MEM_INCLUDED
#define _SWITCH_MEM_INCLUDED

#include <stdio.h>

#ifndef NULL
#define NULL 0
#endif

class CSMessage {
public:
      long int     id;               // Идентификатор сообщения (на графе переходов)
      long int     src;              // Указаиель на источник сообщения (автомат)

      CSMessage(long int  id     = -1,  // Идентификатор сообщения на графе
            long int src       = -1); // Адрес объекта-источника сообщения
};


class CSQueue {                        // Очередь !указателей! на сообщения

      class CSQueueItem {                   // Элемент очереди
      public:
            CSMessage    *msg;              // Экземпляр сообщения
            CSQueueItem *next;              // Следующий элемент очереди
            CSQueueItem(CSMessage *msg);    // Конструктор элемента очереди
      };

public:
      CSQueue();                        // Создание пустой очереди
      ~CSQueue();                       // Уничтожение очереди
      bool IsEmpty();                   // Проверка на пустоту
      bool AddMessage(CSMessage *msg); // Добавление сообщения в очередь
      CSMessage *GetMessage();          // Извлечение сообщения из очереди

protected:
      CSQueueItem *first;               // Первый элемент очереди
      CSQueueItem *last;                // Последний элемент очереди
};


struct CSAutoState {
public:
      int number;
      char *caption;
};


class CSStateArray {
public:
      CSStateArray(int size,            // Размер массива
            int num_arr [],             // Вектор номеров состояний
            char *cap_arr []);          // Вектор названий состояний
      ~CSStateArray();                  // Деструктор
      char *Caption(int state);         // Возврат названия состояния по номеру

private:
      int size;                         // Количество состояний
      CSAutoState *data;                // Собственно массив

      int BinSearch(int state);         // Поиск названия состояния по номеру
};
```

```
class CSAutomate {
public:

	CSAutomate(CSQueue *queue,			// Указатель на очередь сообщений
		const char *automate_id,		// Идентификатор автомата
		CSStateArray *st_arr = NULL,		// Массив идентификаторов состояний
		FILE *f_rec = NULL,			// ОТКРЫТЫЙ файл для записи протокола
		bool recording = false);		// Начальное состояние протоколирования
		~CSAutomate();

	virtual void A(CSMessage &msg) = 0;	// Выбор перехода
	virtual void S() = 0;			// Осуществление перехода

	void StartRec();		// Начать ведение протокола
	void PrintRec();		// Вывод протокола в файл
	void StopRec();			// Остановить ведение протокола
	void Update();			// Обновление внутренних переменных автомата
	int  GetState();		// Интерфейс переменной состояния

protected:
	char *AutomateID;		// Идентификатор автомата
	CSQueue *Queue;			// Указатель на очередь сообщений

	int y_o, y_n;			// Переменные состояния в начале и в конце шага
	int priority;			// Переменная, разрешающая механизм приоритетов
	int action;			// Переменная выбора действия

	CSStateArray *st_arr;		// Массив идентификаторов состояний
	FILE *f_rec;			// Файл для вывода протокола
	bool recording;			// Ведение протокола
};


#endif
```