

Объектно-ориентированный подход к автоматному программированию

Данил Шопырин, Анатолий Шалыто

Предлагается подход к реализации объектно-ориентированных систем с явным выделением состояний. Рассмотрены вопросы повторного использования программных компонентов, параллельных вычислений и автоматического протоколирования. Выполнен обзор классов разработанной библиотеки, предназначенной для реализации автоматной части программы.

Введение

В последнее время в программировании все шире применяются конечные автоматы [1 – 3].

Один из подходов к совместному использованию объектной и автоматной парадигм программирования назван в работах [4–7] «объектно-ориентированное программирование с явным выделением состояний» или «SWITCH-технология» [8]. Хотя в этих работах подробно рассмотрены вопросы проектирования программ этого класса, предложенный в них метод реализации автоматов обладает следующими недостатками:

- не выделено состояние системы в целом;
- в функции, реализующей автомат, применяются два оператора *switch*, так как нет механизма различения действий и деятельности в состояниях. Этот подход снижает удобочитаемость кода и увеличивает вероятность ошибки;
- нет механизма обеспечения повторного использования реализованных автоматов;
- в текст программы функции протоколирования вводятся программистом вручную, что нельзя назвать «автоматическим» построением протокола;
- не предложен механизм обработки ошибок, возникающих при работе системы;
- не предложен механизм организации параллельных вычислений.

В данной работе предлагается подход к реализации программных систем рассматриваемого класса, устраняющий указанные выше недостатки. В качестве базы для разработки «автоматной» части (автоматы, входные и выходные воздействия, инфраструктура) программ с явным выделением состояний предлагается библиотека *STOOL* (SWITCH-Technology Object Oriented Library). Эта библиотека реализована на языке C++ и доступна к загрузке с сайта <http://is.ifmo.ru> (раздел «Проекты»). Остальная часть (контекст) программы разрабатывается традиционным образом.

Определения

Класс автоматов – множество автоматов, реализующих один и тот же граф переходов. Например, классом автоматов является множество одинаковых автоматов, осуществляющих подсчет повторений в последовательности.

Автомат – конкретный экземпляр класса автоматов.

Входное воздействие – булева функция, характеризующая состояние внешней среды. Значение входного воздействия не зависит от порядка и количества обращений к нему. Таким образом, обращение к входному воздействию не изменяет состояния внешней среды.

Выходное воздействие – некоторая операция, изменяющая внешнюю среду.

Система автоматов – совокупность автоматов.

Состояние системы – совокупность состояний всех автоматов системы.

Системный переход – переход системы в другое состояние. Системный переход начинается с запуска некоторого автомата A_i . Этот автомат может запускать другие автоматы. После того как автомат A_i и каждый из запущенных им автоматов совершит не более одного перехода, считается, что системный переход осуществлен.

Исключительная ситуация – возникает в случае невозможности опроса входного воздействия или выполнения выходного воздействия.

Этап автомата – набор последовательных запусков, в течении которых автомат сохраняет свое состояние.

Шаг этапа – каждый запуск автомата в течение этапа.

Действие в состоянии – вызов некоторого выходного воздействия, происходящий во время первого шага этапа автомата.

Деятельность в состоянии – вызов некоторого выходного воздействия, происходящий на каждом шаге этапа.

Действие на переходе – вызов некоторого выходного воздействия, происходящий при переходе автомата из одного состояния в другое. При этом сначала выполняются действия на переходе, а затем изменяется состояние автомата.

Особенности архитектуры программных систем

Предлагаемая архитектура программных систем отличается от предлагаемой в работах [4 – 7]. Эти отличия состоят в следующем:

- автоматы, входные и выходные воздействия являются объектами;
- явно вводятся понятия *класс автоматов* и *экземпляр класса автоматов*;
- каждый автомат не располагает никакой информацией о других автоматах системы. При этом, во-первых, автомат не знает о существовании других автоматов, во-вторых, он не может непосредственно проверять их состояния, как это было предложено, например в работе [8], а в-третьих – не может непосредственно запускать другие автоматы. Единственная связь автомата с «внешним миром» – это входные и выходные воздействия.

Введение понятий *класс автоматов* и *экземпляр класса автоматов* необходимо для обеспечения повторного использования автоматов. Пусть, например, в системе имеются два одинаковых автомата, отличающиеся только входными и выходными воздействиями. При этом нет необходимости каждый из них проектировать отдельно,

а достаточно создать и включить в систему два экземпляра одного и того же класса автоматов, что соответствует парадигме объектно-ориентированного программирования (ООП). Разработанный класс автоматов может повторно использоваться и в других системах.

Введенные ограничения на взаимодействие автоматов повышает их модульность, и тем самым, увеличивают вероятность их повторного использования.

При этом, если необходимо, например, осуществить переход по номеру состояния другого автомата, то вводится входное воздействие, возвращающее значение *true*, если автомат находится в искомом состоянии. Если же необходимо осуществить запуск другого автомата, то вводится выходное воздействие, запускающее этот автомат.

Поясним теперь, почему в рамках предлагаемого подхода понятие «событие» не применяется. Это связано с переносимостью библиотеки STOOOL, на базе которой предлагается реализовывать системы. В разных системах существуют различные модели возникновения и обработки событий, которые весьма трудно обобщаются в понятной и краткой форме. Введение в библиотеку какой-нибудь одной модели обработки событий сузило бы область применения библиотеки, а введение нескольких моделей усложнило бы дизайн библиотеки и простоту ее использования. Поэтому в рамках предлагаемой архитектуры реализация механизмов обработки событий возлагается на пользователя.

Пусть, например, для левой кнопки мыши введено входное воздействие x_0 , возвращающее значение *true*, если кнопка нажата. При получении уведомления от пользовательского интерфейса о том, что кнопка нажата (например, событие *WM_LBUTTONDOWN* в ОС *Windows*), пользователь устанавливает соответствующий флаг и запускает автоматы, обрабатывающие это событие. При получении уведомления о том, что кнопка отпущена, пользователь сбрасывает флаг.

Обзор классов библиотеки STOOOL

Предлагаемый подход базируется на библиотеке STOOOL. Эта библиотека предоставляет абстрактные базовые классы для реализации автоматов, входных и выходных воздействий, а также «инфраструктуру» для организации системы в целом.

При этом для построения конкретного класса автоматов создается потомок абстрактного базового класса *Auto*.

Все классы библиотеки определены внутри пространства имен *stool*.

Опишем классы для реализации автоматов и входных и выходных воздействий:

- *Auto* – базовый класс для разработки классов автоматов. Для определения класса автоматов программист должен переопределить метод *void execution(State& state)*, реализующий граф переходов автомата;
- *State* – класс хранит состояние автомата. Каждый экземпляр класса *Auto* содержит экземпляр класса *State*. Класс *State* обеспечивает протоколирование любого (даже ошибочного) изменения состояния автомата. Изменять экземпляр класса *State* программист может только внутри функции, реализующей граф переходов;
- *Input* – базовый класс для реализации входных воздействий;
- *Output* – базовый класс для реализации выходных воздействий;

- *Impact* – класс описывает процесс выполнения выходного воздействия. Он предоставляет следующие методы: «выполнить», «откатить» и «подтвердить». При каждом выполнении выходного воздействия создается соответствующий этому воздействию экземпляр класса *Impact*, который и осуществляет это воздействие.

Перейдем к описанию классов для создания «инфраструктуры» системы:

- *System* – управляет системой автоматов. Содержит экземпляры классов *ChangeServer*, *AutoEventServer*, которые описаны ниже. Он также хранит список всех автоматов системы. Предполагается, что пользователь может создавать потомков класса *System*;
- *Change* – управляет системным переходом. При возникновении исключительной ситуации во время выполнения системного перехода этот класс отвечает за разворачивание (*unwind*) стека выполненных выходных воздействий. Класс является абстрактным. Библиотека *STOOL* предоставляет два потомка этого класса – *SingleTaskChange* и *MultiTaskChange* (для однопоточной и многопоточной работы). Эти классы могут быть расширены пользователем;
- *ChangeServer* – управляет созданием и уничтожением переходов. Он разработан для использования в однопоточных и в многопоточных системах.

Дальнейшее описание библиотеки приводится в последующих разделах работы. Так например, описываются методы *Output::action()*, *Output::activity()* и *Output::jumpAction()*, реализующие различные типы выходных воздействий.

Применение библиотеки **STOOL**

1. Выделение состояния системы в целом

Пользователь библиотеки *STOOL* может получить список всех автоматов системы, а у каждого автомата – его состояние. Пользователь может также получить информацию о любом выполняющемся в данный момент системном переходе.

Информацию обо всех автоматах системы можно получить с помощью метода *System::enumerate()*, а обо всех выполняющихся переходах – с помощью экземпляра класса *ChangeServer*, возвращаемого методом *System::getChangeServer()*.

Приведем пример кода, распечатывающий список автоматов системы, в котором используется метод *System::enumerate()*:

```
...
struct AutoReceiver
: public ItemsReceiver<const Auto&>
{
    virtual bool receiveCount( int _count ) {
        cout << "всего автоматов: " << _count << endl;
        return true;
    }
    virtual bool receiveItem( int _index, const Auto& _item )
    {
        cout << "автомат #" << _index << ": " << endl;
        cout << "  имя класса автоматов: " << _item.getInfo().getClassName() << endl;
        cout << "  имя автомата: " << _item.getInfo().getInstanceName() << endl;
        cout << "  состояние: " << _item.getInfo().getStateName( _item.getState() )
<< endl;
```

```

        return true;
    }
};
...
system.enumerate( AutoReceiver() );

```

2. Действия и деятельности

Для упрощения шаблона, реализующего автоматы, в котором используются два оператора *switch*, в библиотеке *STOOL* имеются средства для различения действий и деятельности в вершинах.

Деятельности в вершинах реализуется методом *Output::activity()*, а действия – методом *Output::action()*. Объект выходного воздействия сам определяет, находится ли вызывающий автомат в первом шаге этапа. Для выполнения действия на переходе предназначен метод *Output::jumpAction()*.

При описании поведения автомата в виде графа переходов действие в вершине и действие на переходе будем обозначать символом z_i , а деятельность – символом \underline{z}_i .

Рассмотрим граф переходов (рис. 1).

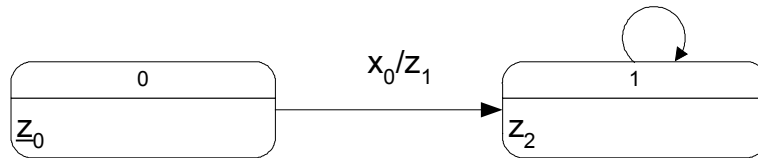


Рис. 1. Граф переходов

Ниже приведена функция, реализующая граф переходов (рис.1) с использованием библиотеки *STOOL*:

```

virtual void execution( State& state ) {
    switch ( state ) {
    case 0:
        io.z0().activity( *this );
        if ( io.x0().is( *this ) ) {
            io.z1().jumpAction( *this );
            state = 1;
        }
        break;
    case 1:
        io.z2().action( *this );
        break;
    }
}

```

В этом примере, действия z_1 и z_2 выполняются не более одного раза.

3. Повторное использование автоматов

Автоматы предлагается реализовывать на основе паттерна, в котором определяется набор необходимых входных (I) и выходных (O) воздействий.

```

class Ai : public Auto {
public:
    struct IO {

```

```

//Входные воздействия
virtual Input& xk() = 0;
virtual Input& xl() = 0;

//Выходные воздействия
virtual Output& zm() = 0;
virtual Output& zn() = 0;
};
private:
    IO& io;

protected:
//Реализация графа перехода
virtual void execution( State& state ) {
    switch ( state ) {
        case 0:
            if ( io.xk().is( *this ) )
                state = 1;
            ...
            break;
        ...
    }
}

public:

//Конструктор
Ai( IO& _io, const string& _instance_name, System& _system )
    : Auto( _instance_name, "Ai", _system )
    , io( _io ) {}
};

```

Такая реализация автомата является системонезависимой, и поэтому появляется возможность повторно использовать классы автоматов.

4. Автоматическое протоколирование

Библиотека *STOOL* предоставляет средства для организации автоматического протоколирования изменений состояний системы. При разработке системы программист должен создать объект, реализующий детали протоколирования (куда записывается протокол, формат протокола и т.д.). Библиотека поддерживает автоматическое протоколирование:

- изменений состояния каждого автомата системы;
- создания и уничтожения автомата;
- начала и конца выполнения автомата;
- опроса входного воздействия;
- выполнения выходного воздействия;
- возникновения исключительной ситуации.

Гарантируется, что объект, реализующий протоколирование, будет уведомлен обо всех вышеуказанных изменениях.

Ниже приведен пример организации протоколирования:

```

class LogSystem : public AutoEventSync {
    int change_number;
public:
    LogSystem( Lockable& _lockable )

```

```

        : AutoEventSync( _lockable )
        , change_number( 0 ) {}

void preamble() {
    cout << ++change_number << ".\t";
}

void onEvent( const Event _event, const AutoEventSync::EventItem& _item ) {
    Lock lock( *this );
    switch ( _event ) {
    case AutoEventSync::E_AFTER_STATE_CHANGED:
        preamble();
        const AutoEventSync::StateEventData& data = _item.getStateEventData();
        const Auto& inst = data.state.getAuto();
        cout << "автомат " << inst.getInfo().getInstanceName()
              << "(" << inst.getInfo().getClassName() << ")"
              << " перешел в состояние: " << data.state
              << " (старое состояние: " << data.old_state << ")"
              << "." << std::endl;
        break;
    }
};

```

5. Механизм обработки ошибок

Для обеспечения устойчивости системы к ошибкам (исключительным ситуациям) вводится следующее ограничение: если не удалось осуществить системный переход, то система остается в исходном состоянии. Таким образом, если причина возникновения исключительной ситуации будет устранена, то при повторном запуске системный переход осуществится так, как если бы исключительной ситуации не возникало. Это аналогично транзакциям в системах баз данных, которые либо выполняются полностью, либо не выполняются вообще.

Для обеспечения устойчивости к ошибкам все выходные воздействия должны иметь возможность отката. Для этого выходные воздействия должны поддерживать следующие три операции:

- «Выполнить». Воздействие выполняется, но при этом сохраняется вся информация, необходимая для отката. Если реализовать откат *сложно*, то захватываются ресурсы, необходимые для выполнения воздействия, а оно само выполняется в операции «Подтвердить».
- «Откатить». Если воздействие было выполнено – производится откат выполненных изменений. Если были захвачены ресурсы, то они освобождаются.
- «Подтвердить». Если воздействие не было выполнено в операции «Выполнить», то оно выполняется. Ресурсы освобождаются. Вероятность возникновения исключительной ситуации во время выполнения этой операции должна быть сведена к минимуму.

Например, необходимо реализовать выходное воздействие, удаляющее некоторый файл. В операции «Выполнить» блокируется доступ к файлу. Если заблокировать файл не удалось (например, он заблокирован другим приложением), то генерируется исключительная ситуация. В операции «Откатить» отменяется блокировка файла, а в операции «Подтвердить» удаляется файл.

Если программист не нуждается в обеспечении устойчивости к ошибкам, то для всех выходных воздействий он может реализовать только операцию «Выполнить».

6. Параллельные вычисления

Для решения многих задач бывает целесообразно организовать несколько потоков выполнения (многопоточные системы). В терминах систем с явным выделением состояний многопоточная система – это система, в которой в один и тот же момент может осуществляться несколько системных переходов.

Основной сложностью при разработке многопоточных систем является организация безопасного использования объекта несколькими потоками.

Библиотека *STOOL* может работать как в однопоточном, так и в многопоточном режиме. Переключение между этими режимами осуществляется передачей конструктору объекта *System* объекта-фабрики [9], создающего объекты *Change*, *ChangeServer* и *Lockable*. В библиотеке реализованы две фабрики:

- класс *Factory* является фабрикой «по умолчанию». Он настраивает библиотеку на однопоточный режим;
- класс *MultiTaskFactory* настраивает библиотеку на многопоточный режим.

Многопоточные объекты (*MultiTaskChange* и *MultiTaskLock*) реализованы с помощью библиотеки *boost::thread* (<http://www.boost.org>).

Любой объект системы, кроме автоматов, может одновременно использоваться только в одном переходе. Пусть выполняются два системных перехода T_0 и T_1 , которые обращаются к входному воздействию x_0 . Процесс разделения объекта x_0 между переходами, осуществляемый средствами библиотеки, показан на рис. 2.

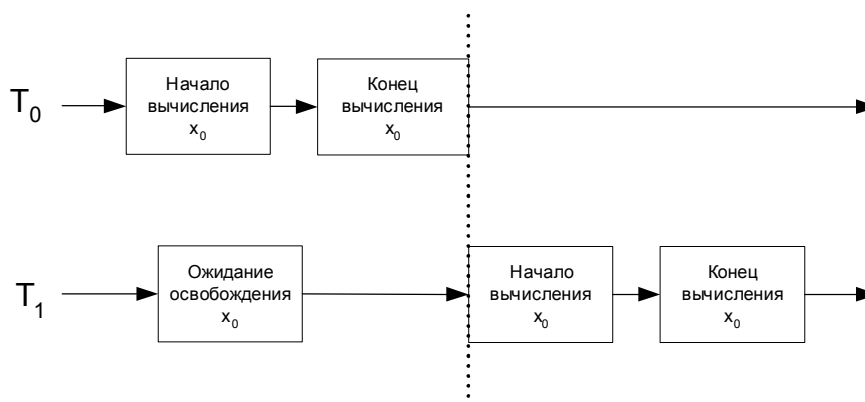


Рис. 2. Разделение объекта между системными переходами

Автоматы являются единственными объектами библиотеки *STOOL*, которые остаются «захваченными» до завершения активизирующего их потока. Если системный переход T_0 запускает некоторый автомат i_0 , то объект, представляющий этот автомат, остается «захваченным» до завершения этого перехода. Процесс разделения автомата i_0 между переходами будет происходить, как показано на рис. 3.

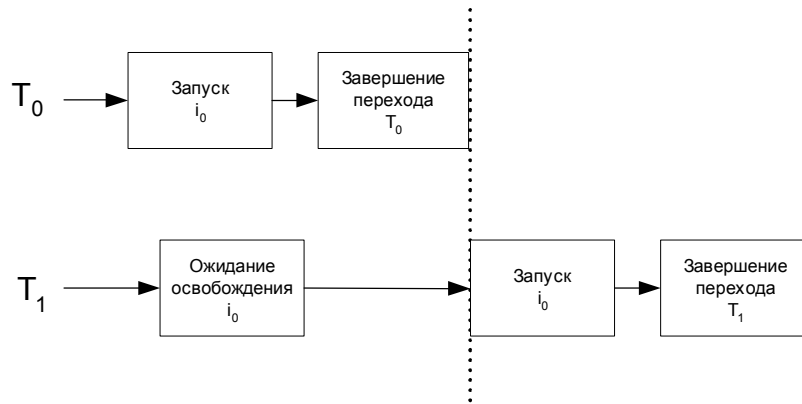


Рис. 3. Разделение автомата между системными переходами

Реализация входных и выходных воздействий

Для реализации входного воздействия создается потомок класса *Input*, и в нем переопределяется метод *bool execution()*:

```

class X0 : public Input {
    Data& data;
protected:
    virtual bool execution() {
        return data.isEnd();
    }
public:
    X0( Data& _data, System& _system )
        : Input( "X0", _system )
        , data( _data ) {}
};
  
```

Для реализации выходного воздействия необходимо сделать два действия:

- определить потомка класса *Impact*;
- определить потомка класса *Output*, который создает экземпляр соответствующего потомка класса *Impact*.

Приведем каркас реализации потомка класса *Impact* и соответствующую реализацию потомка класса *Output*:

```

class IZ0 : public Impact {
public:

    virtual void execute() {
        //Действие
    }

    virtual void rollback(){
        //Откат
    }

    virtual void commit() {
        //Подтверждение
    }
};
  
```

```

class Z0 : public Output {

public:
    virtual Impact* create() {
        return new IZ0();
    }

public:
    Z0( System& _system )
        : Output( "Z0", _system ) {}
};

```

Таким образом, для реализации выходных воздействий программист должен реализовать два класса.

Библиотека *STOOL* предоставляет средства, позволяющие сократить объем кода за счет введения вспомогательных классов.

Для реализации выходных воздействий, запускающих другой автомат, предназначен вспомогательный класс *AutoOutput*, «принимающий» запускаемый автомат в качестве параметра конструктора.

Для того чтобы не создавать два класса для реализации выходных воздействий, предназначены вспомогательные классы *GOutput* и *GOutputPx*. Например, для создания выходного воздействия с некоторым потомком класса *Impact* достаточно следующей строки кода:

```

GOutput<IZ0> z0( "Z0", system );

```

В случае, если входные и выходные воздействия представлены набором функций, для упрощенного создания соответствующих объектов можно воспользоваться вспомогательными функциями *makeFInput* и *makeFOutput*:

```

bool fx0() {
    //Реализация входного воздействия x0
}

void fz0() {
    //Реализация выходного воздействия z0
}

...
Input* x0 = makeFInput( fx0, "x0", system );
Output* z0 = makeFOutput( fz0, "z0", system );

```

Если входные и выходные воздействия представлены методами некоторого класса, то можно воспользоваться функциями *makeFInput* и *makeFOutput* в комбинации с библиотекой *boost::bind*:

```

class Ctx {
public:
    bool fx0() {
        //Реализация входного воздействия x0
        return false;
    }

    void fz0() {
        //Реализация выходного воздействия z0
    }
};

```

```

...
Ctx ctx;
Input* x0 = makeFInput( bind( &Ctx::fx0, ref( ctx ) ), "x0", system );
Output* z0 = makeFOutput( bind( &Ctx::fz0, ref( ctx ) ), "z0", system );

```

В случае, если объекты автоматов, входных и выходных воздействий создаются как члены некоторого класса, то могут быть полезны разработанные макросы:

- *DECLARE_FUNC_INPUT* – объявляет метод, возвращающий экземпляр входного воздействия, которое вызывает некоторую функцию;
- *DECLARE_FUNC_OUTPUT* – объявляет метод, возвращающий экземпляр выходного воздействия, которое вызывает некоторую функцию;
- *DECLARE_AUTO_OUTPUT* – объявляет метод, возвращающий экземпляр выходного воздействия, которое запускает некоторый автомат;
- *DECLARE_AUTO* – объявляет метод, возвращающий экземпляр класса автоматов.

Использование этих макросов позволяет упростить реализацию объектов системы, что будет показано в следующем разделе.

Пример использования библиотеки STOOL

Пусть на вход подается строка символов, завершающаяся нулем. *Словом* назовем непрерывную последовательность не пробельных символов. *Числом* назовем слово, состоящее только из цифр. Необходимо за один проход перевернуть все числа в строке и подсчитать количество слов. Например, при входной строке “test 123”, результирующей строкой будет “test 321”, а количество слов будет равно двум.

Для решения задачи требуется три класса автоматов:

- класс автоматов, управляющих итерированием по строке – A_0 ;
- класс автоматов, переворачивающих числа – A_1 ;
- класс автоматов, подсчитывающих слова – A_2 .

Создадим по одному экземпляру каждого класса автоматов (i_0 , i_1 и i_2). На рис. 4 приведена схема взаимодействия автоматов.

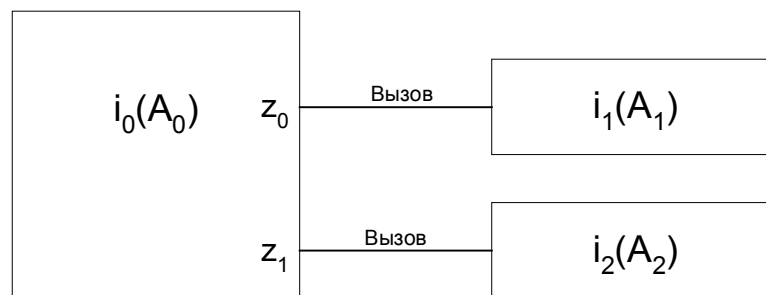


Рис. 4. Схема взаимодействия автоматов

Структурные схемы классов автоматов A_0 , A_1 и A_2 приведены на рис. 5.

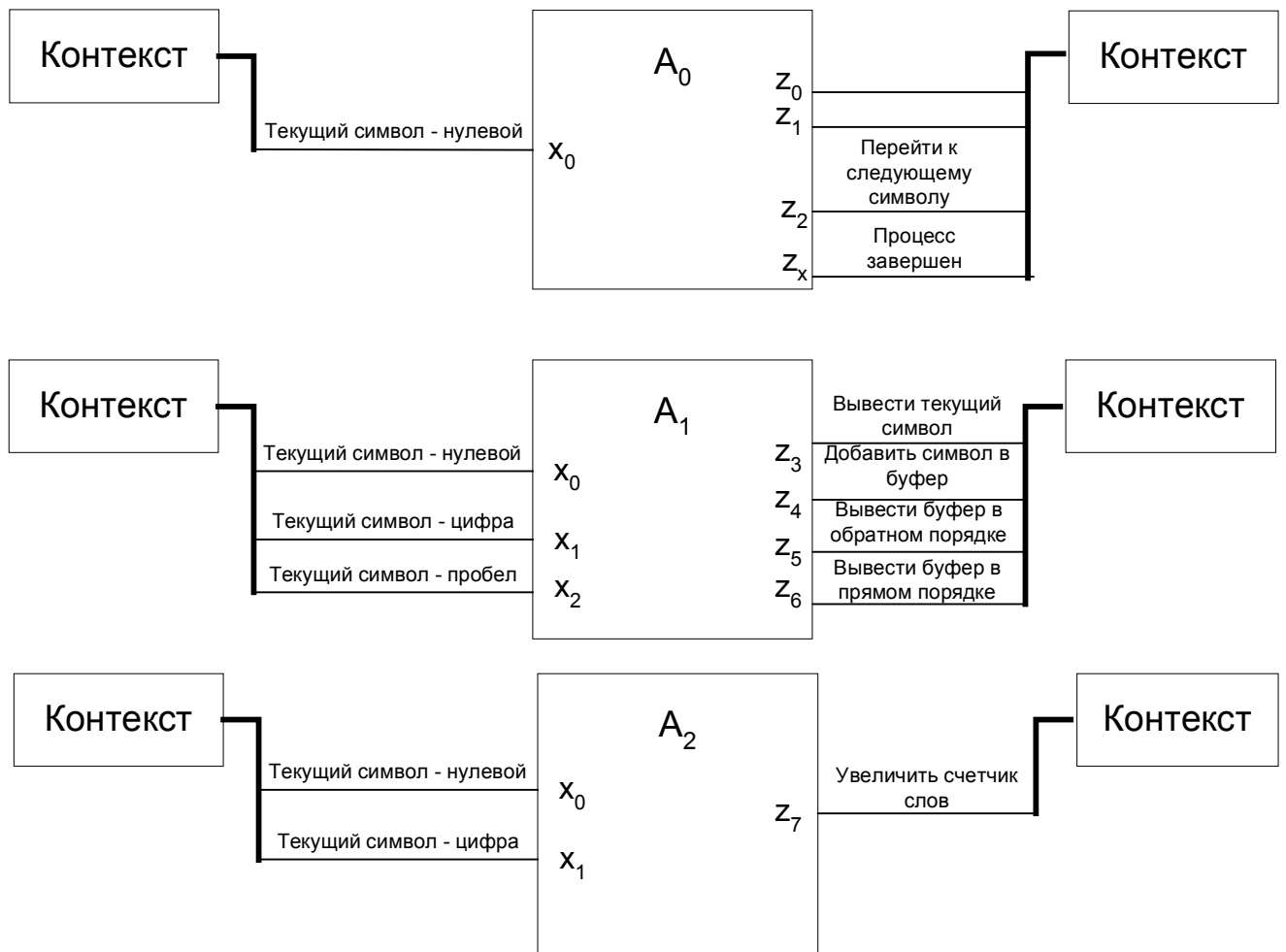


Рис. 5. Структурные схемы классов автоматов A_0 , A_1 и A_2

Графы переходов автоматов A_0 , A_1 и A_2 приведены на рис. 6.

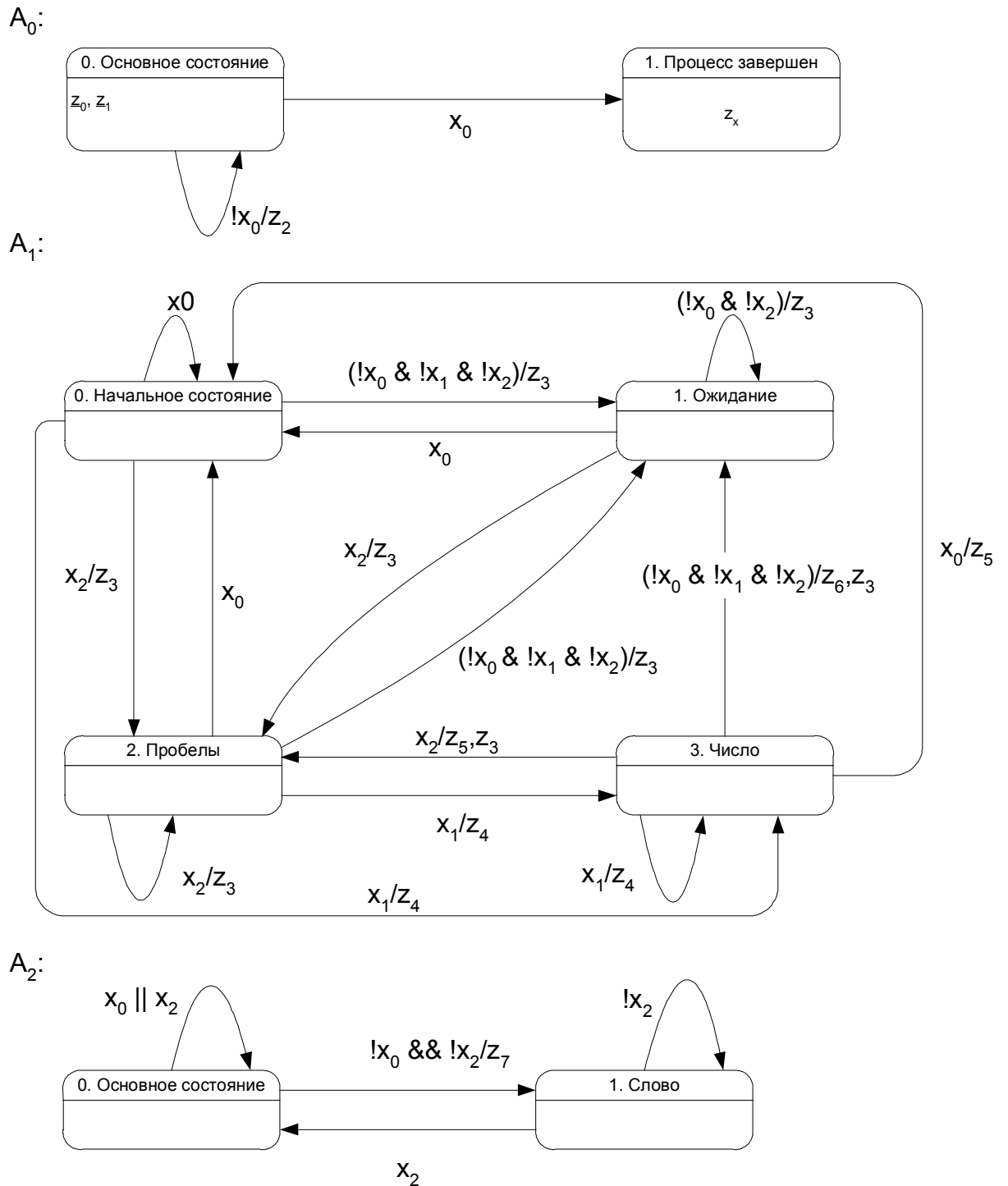


Рис. 6. Графы переходов классов автоматов A_0 , A_1 и A_2

Обратим внимание, что в первом графе переходов использован символ z_x , соответствующий завершению работы системы.

Автоматы A_0 , A_1 и A_2 реализуются на основе изложенного подхода следующим образом:

```
class A0 : public Auto {
public:
    struct IO {
        virtual Input& x0() = 0;
        virtual Output& z0() = 0;
        virtual Output& z1() = 0;
    };
};
```

```

        virtual Output& z2() = 0;
        virtual Output& zx() = 0;
    };
private:
    IO& io;

protected:
    virtual void execution( State& state ) {
        switch ( state ) {
            case 0:
                io.z0().activity( *this );
                io.z1().activity( *this );
                if ( io.x0().is( *this ) )
                    state = 1;
                else if ( !io.x0().is( *this ) )
                    io.z2().jumpAction( *this );
                break;
            case 1:
                io.zx().action( *this );
                break;
        }
    }

public:
    A0( IO& _io, const string& _instance_name, System& _system )
        : Auto( _instance_name, "A0", _system )
        , io( _io ) {}
};

class A1 : public Auto {
public:
    struct IO {
        virtual Input& x0() = 0;
        virtual Input& x1() = 0;
        virtual Input& x2() = 0;
        virtual Output& z3() = 0;
        virtual Output& z4() = 0;
        virtual Output& z5() = 0;
        virtual Output& z6() = 0;
    };
private:
    IO& io;

protected:
    virtual void execution( State& state ) {
        switch ( state ) {
            case 0:
                if ( !io.x0().is( *this ) && !io.x1().is( *this ) && !io.x2().is( *this ) )
                {
                    io.z3().jumpAction( *this );
                    state = 1;
                }
                else if ( io.x1().is( *this ) ) {
                    io.z4().jumpAction( *this );
                    state = 3;
                }
                else if ( io.x2().is( *this ) ) {
                    io.z3().jumpAction( *this );
                    state = 2;
                }
                else if ( io.x0().is( *this ) )
                {}
                break;
            case 1:
                if ( !io.x0().is( *this ) && !io.x2().is( *this ) )
                    io.z3().jumpAction( *this );
                else if ( io.x2().is( *this ) ) {
                    io.z3().jumpAction( *this );
                }
        }
    }
};

```

```

        state = 2;
    } else if ( io.x0().is( *this ) )
        state = 0;
    break;
case 2:
    if ( io.x1().is( *this ) ) {
        io.z4().jumpAction( *this );
        state = 3;
    } else if ( io.x2().is( *this ) )
        io.z3().jumpAction( *this );
    else if ( !io.x0().is( *this ) && !io.x1().is( *this ) && !io.x2().is(
*this ) ) {
        io.z3().jumpAction( *this );;
        state = 1;
    } else if ( io.x0().is( *this ) )
        state = 0;
    break;
case 3:
    if ( io.x1().is( *this ) ) {
        io.z4().jumpAction( *this );
    } else if ( io.x2().is( *this ) ) {
        io.z5().jumpAction( *this );;
        io.z3().jumpAction( *this );;
        state = 2;
    } else if ( io.x0().is( *this ) ) {
        io.z5().jumpAction( *this );;
        state = 0;
    } else if ( !io.x0().is( *this ) && !io.x1().is( *this ) && !io.x2().is(
*this ) ) {
        io.z6().jumpAction( *this );;
        io.z3().jumpAction( *this );;
        state = 1;
    }
    break;
}
}

public:
    A1( IO& _io, const string& _instance_name, System& _system )
        : Auto( _instance_name, "A1", _system )
        , io( _io ) {}
};

class A2 : public Auto {
public:
    struct IO {
        virtual Input& x0() = 0;
        virtual Input& x2() = 0;
        virtual Output& z7() = 0;
    };
private:
    IO& io;

protected:
    virtual void execution( State& state ) {
        switch ( state ) {
        case 0:
            if ( !io.x0().is( *this ) && !io.x2().is( *this ) ) {
                io.z7().jumpAction( *this );
                state = 1;
            } else if ( io.x0().is( *this ) || io.x2().is( *this ) )
                {}
            break;
        case 1:
            if ( io.x2().is( *this ) ) {

```

```

        state = 0;
    }
}

public:
    A2( IO& _io, const string& _instance_name, System& _system )
        : Auto( _instance_name, "A2", _system )
        , io( _io ) {}
};

```

Приведенная реализация классов автоматов изоморфна графам переходов.

Завершив построение автоматной части программы, перейдем к реализации ее «контекста», реализующего такие функции, как например, проверка на конец строки:

```

class Data {
    string in;
    string out;
    string buffer;
    string::iterator cursor;
    int nwords;

public:
    Data( const string& _in )
        : in( _in )
        , nwords( 0 )
        , cursor( in.begin() ) {}
    bool isEnd() { return in.end() == cursor; }
    bool isDigit() { return isdigit( *cursor ) != 0; }
    bool isSpace() { return isspace( *cursor ) != 0; }
    void forward() { cursor++; }
    void back() { cursor--; }
    void output() { out += *cursor; }
    void push() { buffer += *cursor; }
    void outBuffer() { out = out + buffer; buffer.clear(); }
    void outBufRev() { reverse( buffer.begin(), buffer.end() ); outBuffer(); }
    void incWords() { nwords++; }
    string getOutput() const { return out; }
    int getWordsCount() const { return nwords; }
};

```

Этот класс не содержит никакой «логики».

Теперь рассмотрим класс *MySystem* (потомок класса *stool::System*) – часть инфраструктуры системы. Этот класс создает и связывает между собой автоматы, входные и выходные воздействия, а также контекст:

```

class MySystem : public System, public A0::IO, public A1::IO, public A2::IO {
    Data data;
    bool stopped;

    DECLARE_AUTO(A0, i0, *this, *this);
    DECLARE_AUTO(A1, i1, *this, *this);
    DECLARE_AUTO(A2, i2, *this, *this);

    DECLARE_FUNC_INPUT ( x0, bind( &Data::isEnd,      ref( data) ), *this );
    DECLARE_FUNC_INPUT ( x1, bind( &Data::isDigit,    ref( data) ), *this );
    DECLARE_FUNC_INPUT ( x2, bind( &Data::isSpace,    ref( data) ), *this );
    DECLARE_AUTO_OUTPUT( z0, i1(), *this );
    DECLARE_AUTO_OUTPUT( z1, i2(), *this );
    DECLARE_FUNC_OUTPUT( z2, bind( &Data::forward,    ref( data) ), *this );
    DECLARE_FUNC_OUTPUT( z3, bind( &Data::output,     ref( data) ), *this );
};

```



```

    DECLARE_FUNC_OUTPUT( z4, bind( &Data::push,          ref( data) ), *this );
    DECLARE_FUNC_OUTPUT( z5, bind( &Data::outBufRev, ref( data) ), *this );
    DECLARE_FUNC_OUTPUT( z6, bind( &Data::outBuffer, ref( data) ), *this );
    DECLARE_FUNC_OUTPUT( z7, bind( &Data::incWords, ref( data) ), *this );
    DECLARE_FUNC_OUTPUT( zx, bind( &MySystem::stop, ref( *this) ), *this );
protected:
    virtual void stop() { stopped = true; }
public:
    MySystem( const string& _in, Factory& _factory )
        : System( _factory )
        , data( _in )
        , stopped( false ) {}

    void run() {
        while ( !stopped )
            getChangeServer().start( i0() );
    }

    const Data& getData() const { return data; }
};

```

В инфраструктуру входит также объект, реализующий протоколирование:

```

class LogSystem : public AutoEventSync {
    int change_number;
    ostream& stream;
public:
    LogSystem( Lockable& _lockable, ostream& _stream )
        : AutoEventSync( _lockable )
        , change_number( 0 )
        , stream( _stream ) {}

    void preamble() {
        stream << ++change_number << ".\t";
    }

    void onEvent( const Event _event, const AutoEventSync::EventItem& _item ) {
        Lock lock( *this );
        switch ( _event ) {
            case AutoEventSync::E_AFTER_STATE_CHANGED:
                preamble();
                stream << "автомат "
                    << _item.getStateEventData().state.getAuto().getInfo().getInstanceName()
                    << " ("
                    << _item.getStateEventData().state.getAuto().getInfo().getClassName()
                    << ")" << " перешел в состояние: " << _item.getStateEventData().state
                    << " (старое состояние: " << _item.getStateEventData().old_state << ")"
                    << "." << std::endl;
                break;
            case AutoEventSync::E_AFTER_INPUT_CHECKED:
                preamble();
                stream << "опрошено входное воздействие "
                    << _item.getInputResultEventData().input.getName()
                    << ", результат: " << _item.getInputResultEventData().result
                    << "." << std::endl;
                break;
            case AutoEventSync::E_AFTER_OUTPUT_ACTIVATED:
                preamble();
                stream << "выполнено выходное воздействие "
                    << _item.getOutputEventData().output.getName() << "." << std::endl;
                break;
            case AutoEventSync::E_AFTER_EXCEPTION:
                preamble();
                std::exception* ex = _item.getExceptionEventData().exception;

```

```

        if ( ex )
            stream << "возникло исключение: " << ex->what();
        else
            stream << "возникло неизвестное исключение";
        stream << std::endl;
        break;
    }
}
};

```

При этом, функция *main* реализуется следующим образом:

```

int main()
{
    const int INPUT_SIZE = 256;
    char input[ INPUT_SIZE + 1 ];
    cout << "input string: ";
    cin.getline( input, INPUT_SIZE );
    Factory factory;
    ofstream log_stream("log.txt");
    LogSystem log( factory.makeLockable(), log_stream );
    MySystem system( input, factory );
    system.getAutoEventService().inject( log );
    system.run();
    cout << "output: " << system.getData().getOutput() << endl;
    cout << "words count: " << system.getData().getWordsCount() << endl;
    cout << "press any key...";
    _getch();
    return 0;
}

```

Как следует из рассмотренного примера, его реализация практически не содержит никакой «лишней информации» (кроме протоколирования). В реальных системах объем кода, обеспечивающего протоколирование, существенно меньше по сравнению с размером кода остальной части системы.

В заключение отметим, что предложенный подход позволил устранить недостатки SWITCH-технологии, перечисленные во введении.

Литература

1. Сацкий С. Дизайн шаблона конечного автомата на C++ //RSDN Magazine. 2003. № 1.
2. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. М.: ДМК, 2000.
3. Любченко В.С. О бильярде с Microsoft Visual C++ 5.0 //Мир ПК. 1998. № 1.
4. Шалыто А.А., Туккель Н.И. Танки и автоматы. //BYTE/Россия. 2003. №2. <http://is.ifmo.ru> (раздел «Статьи»).
5. Туккель Н.И., Шалыто А.А. Система управления танком для игры Robocode. Объектно-ориентированное программирование с явным выделением состояний. <http://is.ifmo.ru> (раздел «Проекты»).
6. Шалыто А.А., Туккель Н.И. Программирование с явным выделением состояний // Мир ПК. 2001. № 8, 9. <http://is.ifmo.ru> (раздел «Статьи»).

7. *Шалыто А.А., Туккель Н.И.* Реализация автоматов при программировании событийных систем. //Программист. 2002. № 4. <http://is.ifmo.ru> (раздел «Статьи»).
8. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
9. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.