Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

В.С. Гуров, А.А. Шалыто

Построение простого клиент-серверного приложения на основе автоматного подхода
(ICQ и автоматы)

Проектная документация

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

Санкт-Петербург
2004

Оглавление

Введение

В работе [1] предложен подход, названный «объектно-ориентированное программирование с явным выделением состояний», эффективность которого показана на примере разработки событийной системы управления «танком». Особенность этого подхода состоит в том, что поведение объектов описывается конечными автоматами, представляемыми в форме графов переходов, которые реализуются с помощью оператора *SWITCH* языка *C*.

В работе [2] разработан подход, позволяющий автоматизировать процесс получения исполняемой программы по описанию схем связей и графов переходов автоматов. Этот подход состоит из следующих этапов:

- на основе анализа предметной области строиться диаграмма классов системы, выделяются объекты и управляющие ими автоматы;
- для каждого автомата, используя свободно распространяемую среду *ArgoUML* [3], предназначенную для разработки диаграмм на языке *UML*, с помощью нотации диаграммы классов строится его схема связей. На этой схеме справа отображаются объекты управления, а слева - автоматы, в том числе и вложенные;
- связи между объектами управления и автоматами именуются символами Oi, соответствующими указанным объектам;
- каждый объект управления содержит два типа основных методов, которые реализуют входные переменные (Xj) и выходные воздействия (Zk);
- для каждого автомата в указанной среде разработки строится граф переходов типа *Мура-Мили*, в котором на дугах кроме входных переменных указываются также события (Ev);
- состояния на графе переходов могут быть простыми и сложными. Если в состояние вложено другое состояние, то оно называется сложным. В противном случае состояние является простым. Все сложные состояния неустойчивы, а все простые - устойчивы;
- в отличие от традиционной SWITCH-технологии, при использовании сложных состояний в автомате при появлении события может выполняться более одного перехода. Это происходит в связи с тем, что сложное состояние является неустойчивым и автомат осуществляет переходы до тех пор, пока не достигнет первого из простых (устойчивых) состояний;
- диаграмма классов и графы переходов автоматически преобразуются в текстовое описание конечного автомата в формате *XML* с помощью разработанного конвертера;
- каждая входная переменная и каждое выходное воздействие реализуется вручную в соответствии с его функциональностью.

На языке *Java* разработана среда исполнения (интерпретатор) полученного *XML*-описания. При этом сначала описание однократно и целиком преобразуется в соответствующее внутреннее объектное представление конечных автоматов. В результате образуется система, состоящая из среды исполнения, объектного представления автоматов и объектов управления (функций, реализующих входные переменные и выходные воздействия).

Эта система при появлении каждого события:

- анализирует его и входные переменные;
- выполняет выходные воздействия;

- запускает вложенные автоматы.

При этом, как при работе, так и при отладке, могут вестись два типа логов – длинный и короткий. Первый из них отражает внутреннее поведение автоматов, в том числе вложенных, записи для которых выделяются отступами. Во втором логе отражаются только входные переменные и выходные воздействия.

В настоящей работе описанный подход иллюстрируется на примере простого клиент-серверного приложения.

1. Постановка задачи

Требуется создать систему мгновенного обмена сообщениями между любым количеством пользователей (простой аналог ICQ – «Я Ищу Тебя»). Система состоит из сервера сообщений и однотипных клиентских приложений. После запуска клиентское приложение присоединяется к серверу сообщений. Затем оно может получить список пользователей уже присоединенных к серверу и обмениваться сообщениями с этими пользователями.

При выходе какого-либо пользователя из системы, остальные пользователи должны быть уведомлены об этом. При завершении работы сервера, все подсоединенные пользователи должны отключится от него.

При взаимодействии между серверным и клиентским приложениями в качестве транспортного протокола должен использоваться протокол *TCP*. Этот протокол позволяет пересылать массивы байт. Для обмена сообщениями на более высоком (прикладном) уровне должен быть разработан другой протокол.

Будем говорить, что клиентское и серверное приложения обмениваются ***сообщениями***, в то время как пользователи системы обмениваются ***репликами***.

2. Диаграмма классов

Диаграмма классов системы приведена на рис. 1, 2.

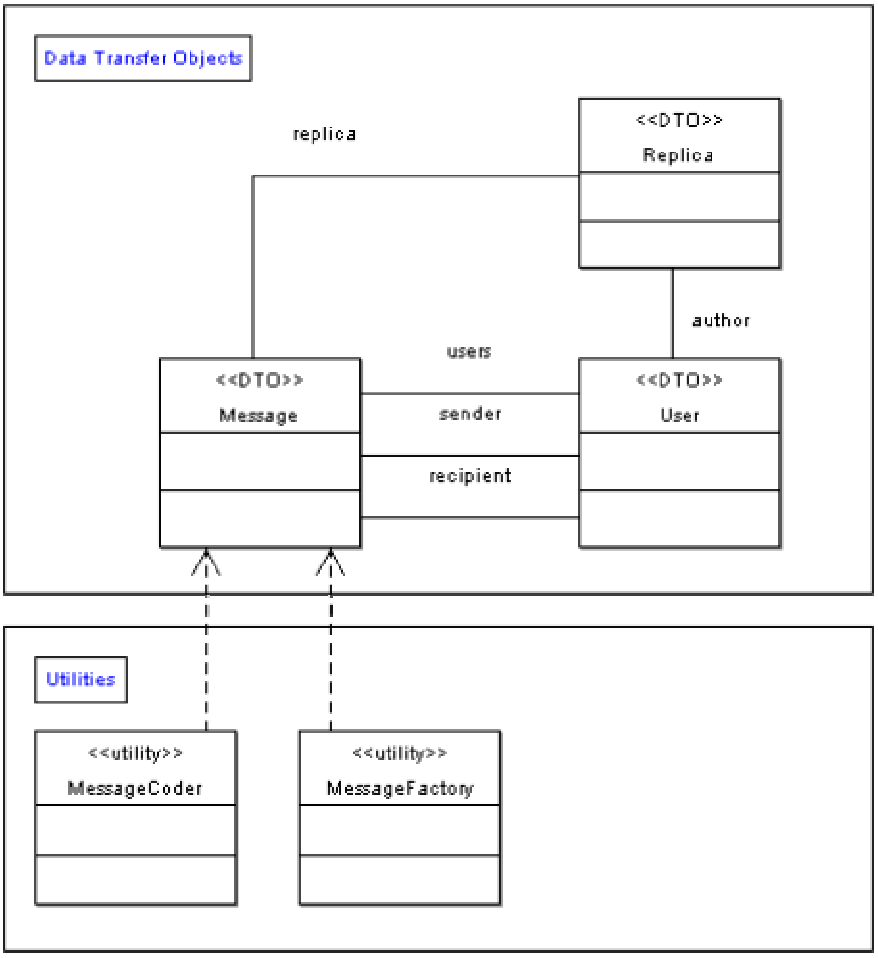

Рис. 1. Диаграмма классов. Часть 1

Диаграмма на рис. 1 содержит две группы классов:

- группа классов для передачи данных (*Data Transfer Objects*). Эти классы не обладают поведением и используются только как контейнеры данных;
- группа вспомогательных классов (*Utilities*). Они реализуют утилитные функции;

Диаграмма на рис. 2 содержит три группы классов:

- классы, являющиеся частью среды исполнения (*State Machine Framework Classes*);
- классы, являющиеся «входными точками» для серверного и клиентского приложений (*Applications*);
- объекты управления (*Controllable Objects*).

Опишем каждую из этих групп. Группа классов для передачи данных состоит из следующих элементов:

- класс *User* содержит информацию о пользователе системы. Сервер также может выступать в роли пользователя – инициатора сообщений;
- класс *Replica* содержит реплики, которыми пользователи обмениваются друг с другом. Реплика всегда имеет автора;

- класс *Message* содержит сообщения, которыми обмениваются клиентское и серверное приложения. Эти сообщения всегда имеют отправителя и получателя. Сообщения (в зависимости от типа) могут содержать или не содержать реплику или список присоединенных пользователей. Более подробное описание данного класса приведено в следующем разделе.

Группа вспомогательных классов содержит в себе:

- класс *MessageCoder*, который реализует преобразование сообщений в массив байт и обратное преобразование, обеспечивая тем самым взаимодействие между прикладным протоколом обмена сообщениями и низкоуровневым транспортным протоколом *TCP*;
- класс *MessageFactory*, который реализует фабрику [4] сообщений. Он позволяет создавать сообщения всех необходимых типов.

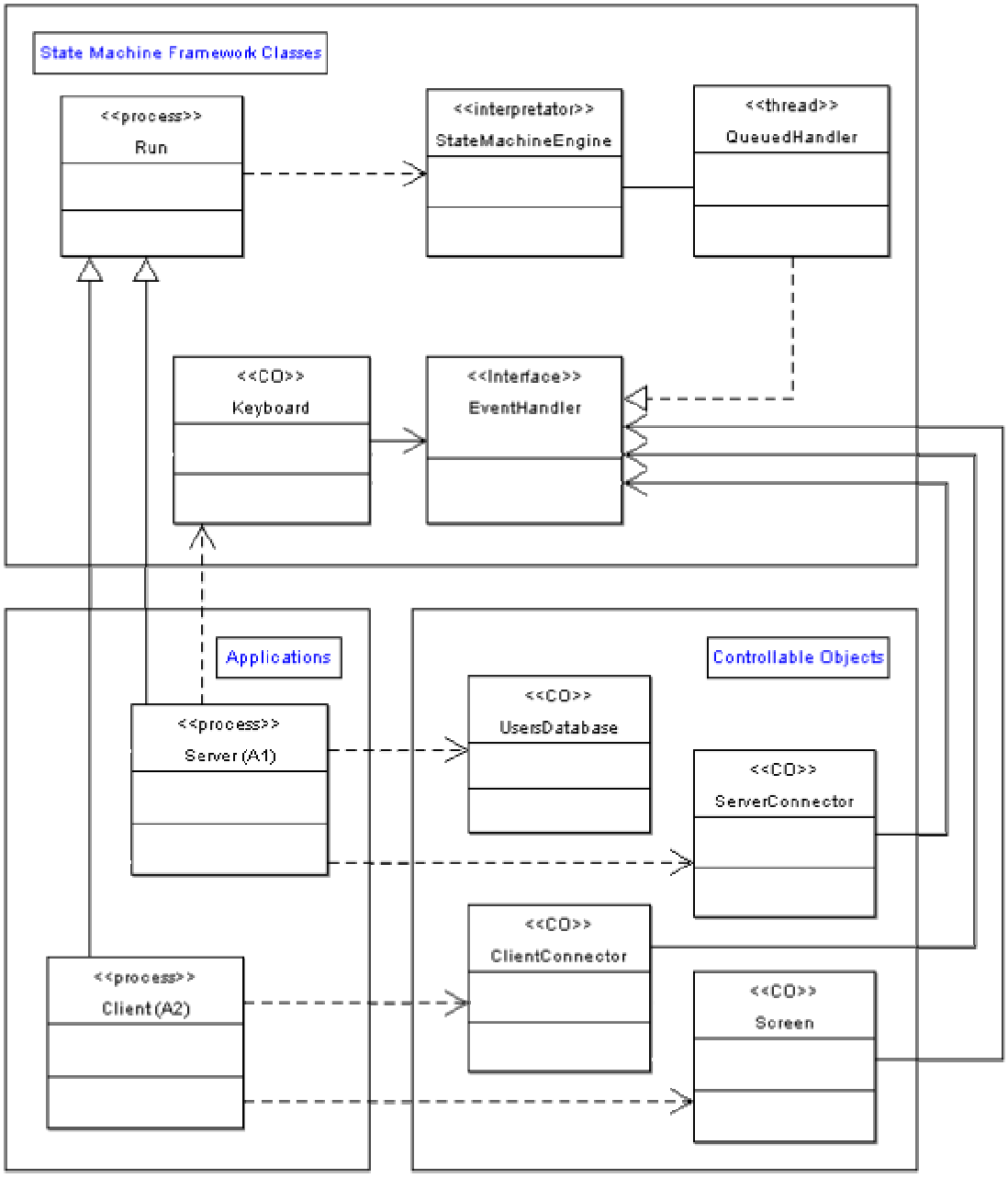

Рис. 2. Диаграмма классов. Часть 2

Классы, являющиеся частью среды исполнения:

- класс *Run* преобразует *XML*-описание системы взаимосвязанных автоматов во внутреннее объектное представление и запускает интерпретатор;
- класс *StateMachineEngine* является интерпретатором объектного представления конечных автоматов;
- класс *QueuedHandler* реализует очередь событий. По мере поступления событий этот класс передает их на обработку интерпретатору;
- класс *Keyboard* реализует объект управления клавиатурой.

Классы *Server* и *Client* наследуются от класса *Run* и расширяют его функциональность, за счет дополнительной настройки серверного и клиентского приложений при старте.

Классы объектов управления:

- класс *ServerConnector* реализует объект управления, предоставляющий клиентам интерфейс для взаимодействия с сервером сообщений;
- класс *UserDatabase* реализует объект управления, представляющий базу данных зарегистрированных пользователей;
- класс *ClientConnector* реализует объект управления, отвечающий за взаимодействие клиента с сервером сообщений;
- класс *Screen* реализует объект управления, который предоставляет возможность управлять графическим интерфейсом пользователя.

3. Протокол взаимодействия клиентского и серверного приложений

Для того чтобы спроектировать протокол обмена сообщениями сначала необходимо определить требования к нему, а также описать сценарии его работы (прецеденты).

Перечислим требования к протоколу:

- возможность авторизованного входа пользователей в систему;
- возможность обмена репликами между пользователями;
- возможность получения по запросу списка других присоединенных пользователей.

Протокол должен обеспечивать выполнение следующих сценариев:

1. Вход в систему. Клиентское приложение посылает сообщение серверу, содержащее имя и пароль пользователя. Сервер проверяет правильность этих параметров на соответствие внутренней базе данных о пользователях. После этого он посылает клиенту либо подтверждение о входе, либо отказ. При подтверждении входа все уже присоединенные клиенты уведомляются о появлении нового пользователя.
2. Обмен репликами. Клиент посылает сообщение серверу, содержащее имя получателя и реплику. Сервер пересылает сообщение адресату.
3. Запрос списка пользователей. Клиент посылает сообщение с запросом списка пользователей. Сервер формирует этот список и посылает его обратно клиенту.
4. Выход клиента из системы. Клиент посылает уведомление серверу о своем выходе. Сервер уведомляет всех остальных присоединенных клиентов об этом событии.
5. Завершение работы сервера. Сервер рассылает всем присоединенным клиентам уведомление о завершении своей работы. Все клиенты переходят в состояние «Отсоединен».

Сообщения, которыми обмениваются клиент и сервер, можно разделить на несколько типов, которые описаны в табл. 1.

Таблица 1. Типы сообщений

Тип сообщения	Инициатор сообщения	Описание
LOGIN	Клиент	Запрос входа в систему. Сообщение должно содержать имя и пароль пользователя
LOGIN	Сервер	Подтверждение входа
LOGOUT	Клиент	Сообщение о выходе из системы
LOGOUT	Сервер	Отказ во входе в систему либо уведомление о завершении работы сервера
REPLICA	Клиент	Посылка реплики пользователем другому пользователю
REPLICA	Сервер	Пересылка сервером реплики адресату
USERS	Клиент	Запрос списка присоединенных пользователей
USERS	Сервер	Сообщение, содержащее список присоединенных пользователей

4. Поведение серверного приложения

Схема связей автомата, реализующего поведение серверного приложения, приведена на рис. 3.

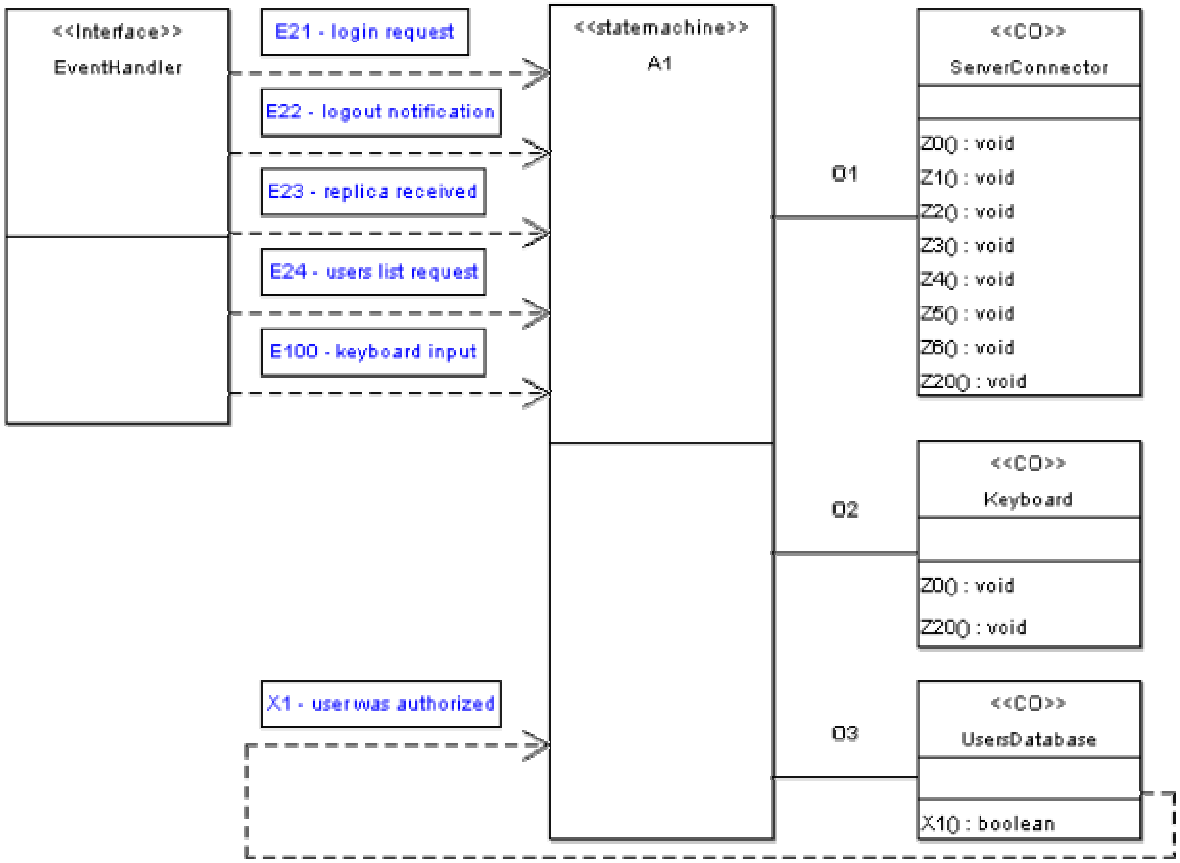

Рис. 3. Схема связей серверного автомата

Для реализации входных переменных и выходных воздействий необходимо знать их функциональность и данные, которые они получают или заносят в общий поток [5, 6]. В данной работе этот поток управляется графом переходов. При этом отметим, что в дальнейшем будем различать входные переменные и входные данные, выходные воздействия и выходные данные.

Поясним, как выполняется управление потоком данных. Модель поведения серверного приложения является событийной. Поэтому первоначальный источник данных – событие. Далее потребителями и поставщиками данных могут быть только входные переменные и выходные воздействия. Поток начинается с выходных данных пришедшего события и завершается входными данными последнего воздействия, выполненного автоматом при обработке события.

Общее *правило для формирования корректного потока данных* можно сформулировать следующим образом: если для каждого перехода сформировать последовательность из события, входных переменных и выходных воздействий, записанных на переходе, то

множество входных данных каждого элемента последовательности должно быть не шире, чем объединение множеств выходных данных всех предшествующих элементов. Формальное описание этого правила приведено в Приложении. Это правило является ограничением при проектировании.

На рис. 3 выделены три объекта управления:

- объект *ServerConnector* (O1) предоставляет клиентам интерфейс для взаимодействия с сервером сообщений;
- объект *Keyboard* (O2) позволяет взаимодействовать с устройством ввода;
- объект *UserDatabase* (O3) реализует базу данных зарегистрированных пользователей.

Отметим, что класс *A1* в программной реализации отсутствует. Он изображается на схеме связей для того, чтобы сгенерировать *XML*-описание.

В табл. 2 приведены описания событий, а в табл. 3 – описания входных переменных и выходных воздействий серверного приложения.

Таблица 2. События для серверного приложения

Событие	Описание	Аргументы (выходные данные) в формате имя:тип
E21	Запрос входа в систему	MESSAGE: Message
E22	Уведомление о выходе из системы	MESSAGE: Message
E23	Получен текст	MESSAGE: Message
E24	Запрос на получение списка пользователей	USER: User
E100	Введена строка с клавиатуры	INPUT: String

Таблица 3. Входные переменные и выходные воздействия серверного приложения

Воздействия	Описание	Входные данные	Выходные данные
Connector (O1)			
Z0	Инициализация объекта управления		
Z1	Послать сообщение, подтверждающее разрешение входа в систему	MESSAGE: Message	
Z2	Послать сообщение, запрещающее вход в систему	MESSAGE: Message REASON: String	
Z3	Переслать текст всем присоединенным пользователям	MESSAGE: Message	
Z4	Послать список пользователей инициатору сообщения	MESSAGE: Message	
Z5	Послать список пользователей всем присоединенным пользователям		
Z6	Закрыть соединение с клиентом	MESSAGE: Message	
Z20	Послать всем пользователям уведомление о завершении работы сервера и закрыть соединение со всеми пользователями		
Keyboard (O2)			
Z0	Инициализация объекта управления		
Z20	Перестать обрабатывать сообщения от клавиатуры		
UserDatabase (O3)			
X1	Провести авторизацию пользователя. Если авторизация не удалась, то возвращает причину ошибки	MESSAGE: Message	REASON: String

Используя эти обозначения, построим граф переходов для сервера (рис. 4).

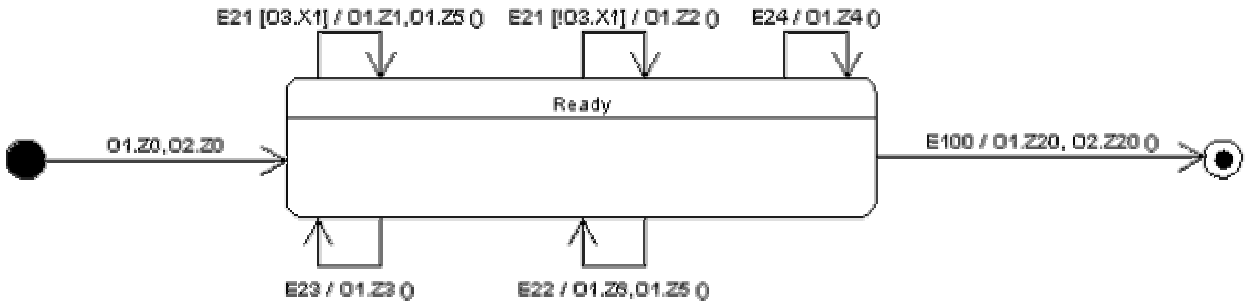

Рис. 4. Граф переходов серверного автомата A1

Можно показать, что все переходы этого графа удовлетворяют правилу построения потока данных. Например, для дуги, помеченной выражением *E21[!O3.X1]/O1.Z2*, это правило справедливо:

E21 (out Message) -> O3.X1 (in Message, out Reason) -> O1.Z2 (in Message, in Reason)

Обратим внимание, что автомат A1 является формальным описанием протокола взаимодействия клиента и сервера. Он имеет три состояния, из которых только одно рабочее. Если бы разрабатывался протокол с большим количеством рабочих состояний, то пришлось бы на серверной стороне иметь по одному автомату для каждого присоединенного клиента. Это объясняется тем, что в любой момент времени каждый из присоединенных клиентов может находиться в состоянии, отличном от другого.

При этом можно говорить не об отдельном автомате для каждого присоединенного клиента, а об отдельном хранении конфигурации автомата для каждого клиента. В этом случае при появлении события от клиента серверному автомату помимо этого события передается также и состояние клиента.

С помощью разработанного конвертора [7] на основе диаграмм на рис. 3, 4 автоматически строится *XML*-описание автомата A1 (Листинг 1). Данное описание изоморфно указанным диаграммам.

Листинг 1. XML-описание серверного автомата A1

```
<?xml version="1.0" encoding="UTF-8"?>
<stateMachine name="A1">

    <!—Связь с объектами управления получена на основе диаграммы связей автомата -->
    <controlledObject name="O1">

<controlledObjectClassName>org.ifmo.messenger.server.co.ServerConnector</controlledObjectClassName>
    </controlledObject>
    <controlledObject name="O2">
        <controlledObjectClassName>org.ifmo.messenger.server.co.Keyboard</controlledObjectClassName>
    </controlledObject>
    <controlledObject name="O3">

<controlledObjectClassName>org.ifmo.messenger.server.co.UserDatabase</controlledObjectClassName>
    </controlledObject>

    <!— Все состояния автомата -->
    <state name="TOP" type="0">
        <state name="Ready" type="0"/>
        <state name="EndState1" type="2"/>
        <state name="StartState1" type="1"/>
    </state>
```

```
    <!– Все переходы автомата -->
    <transition>
        <sourceStateRef>Ready</sourceStateRef>
        <targetStateRef>EndState1</targetStateRef>
        <event name="E100"/>
        <outcomeEffect>O1.Z20</outcomeEffect>
        <outcomeEffect>O2.Z20</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>StartState1</sourceStateRef>
        <targetStateRef>Ready</targetStateRef>
        <outcomeEffect>O1.Z0</outcomeEffect>
        <outcomeEffect>O2.Z0</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Ready</sourceStateRef>
        <targetStateRef>Ready</targetStateRef>
        <event name="E21"/>
        <condition>
            <expression>!O3.X1</expression>
        </condition>
        <outcomeEffect>O1.Z2</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Ready</sourceStateRef>
        <targetStateRef>Ready</targetStateRef>
        <event name="E24"/>
        <outcomeEffect>O1.Z4</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Ready</sourceStateRef>
        <targetStateRef>Ready</targetStateRef>
        <event name="E23"/>
        <outcomeEffect>O1.Z3</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Ready</sourceStateRef>
        <targetStateRef>Ready</targetStateRef>
        <event name="E21"/>
        <condition>
            <expression>O3.X1</expression>
        </condition>
        <outcomeEffect>O1.Z1</outcomeEffect>
        <outcomeEffect>O1.Z5</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Ready</sourceStateRef>
        <targetStateRef>Ready</targetStateRef>
        <event name="E22"/>
        <outcomeEffect>O1.Z6</outcomeEffect>
        <outcomeEffect>O1.Z5</outcomeEffect>
    </transition>
</stateMachine>
```

Этот листинг является описанием **поведения** серверного приложения. Поэтому о нем, можно говорить, как об **автоматной программе**. После десятиминутного разговора об этом **J** авторы пришли к выводу, что описание само по себе все-таки не является программой, так как требует дополнительно реализации входных переменных и выходных воздействий.

В силу изоморфизма и благодаря автоматической генерации описания, диаграммы, изображенные на рис. 3, 4, можно считать каркасом, описывающим логику программы.

5. Поведение клиентского приложения

Окно пользовательского интерфейса представлено на рис. 5. Ссылками показаны события, порождаемые кнопками.

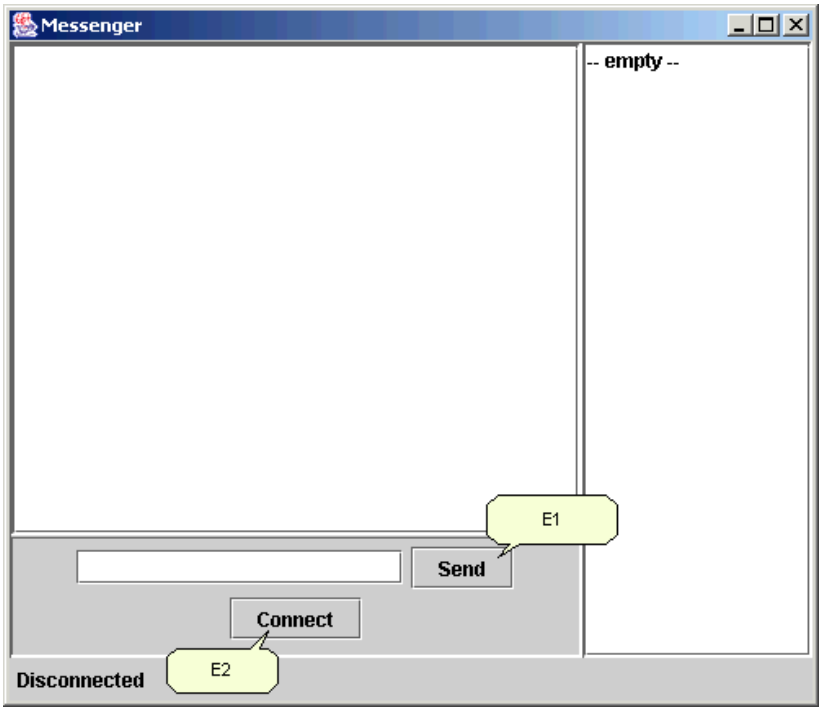

Рис. 5. Пользовательский интерфейс клиентского приложения

Схема связей автомата A2, реализующего поведение клиентского приложения, представлена на рис. 6.

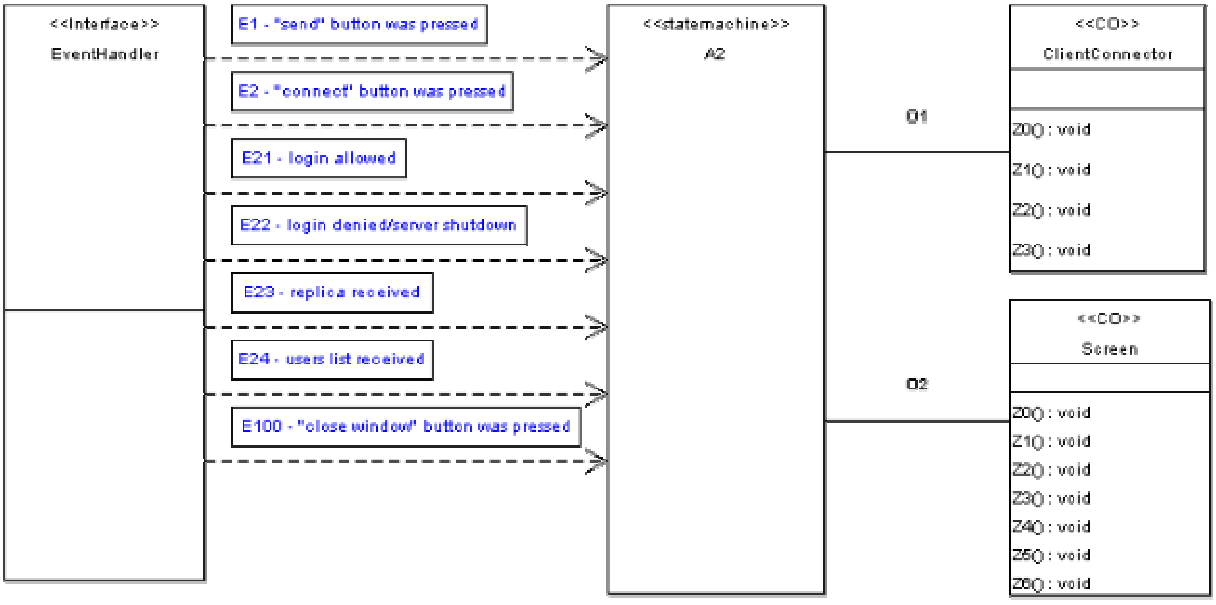

Рис. 6. Схема связей клиентского автомата

Автомат A2 воздействует на следующие объекты управления:

- объект *Connector* (O1), отвечающий за взаимодействие с сервером сообщений;
- объект *Screen* (O2), обеспечивающий возможность управления графическим интерфейсом пользователя.

В табл. 4 приведено описание событий, воздействующих на автомат A2, а в табл. 5 – описание выходных воздействий автомата A2.

Таблица 4. События для клиентского приложения

Событие	Описание	Аргументы (выходные данные) в формате имя:тип
E1	Нажата кнопка *Send*	REPLICA: String
E2	Нажата кнопка *Connect*	
E21	Получено сообщение, разрешающее вход в систему	
E22	Получено сообщение, запрещающее вход в систему или уведомление о выходе сервера из системы	REASON: String
E23	Получен текст	REPLICA: Replica
E24	Получен список пользователей	USERS: User []
E100	Запрос на закрытие окна	

Таблица 5. Выходные воздействия клиентского приложения

Воздействия	Описание	Входные данные	Выходные данные
Connector (O1)			
Z1	Присоединится к серверу		
Z2	Отсоединится от сервера		
Z3	Послать текст всем пользователям	REPLICA: String	
Screen (O2)			
Z1	Показать окно		
Z2	Спрятать окно		
Z3	Обновить список подключенных пользователей	USERS: User []	
Z4	Отобразить текст сообщения	REPLICA: Replica	
Z5	Изменить надпись *Connect* на *Disconnect* на кнопке. Обновить строку состояния окна		
Z6	Изменить надпись *Disconnect* на *Connect* на кнопке. Обновить строку состояния окна	REASON: String	

На рис. 7 показан граф переходов автомата A2.

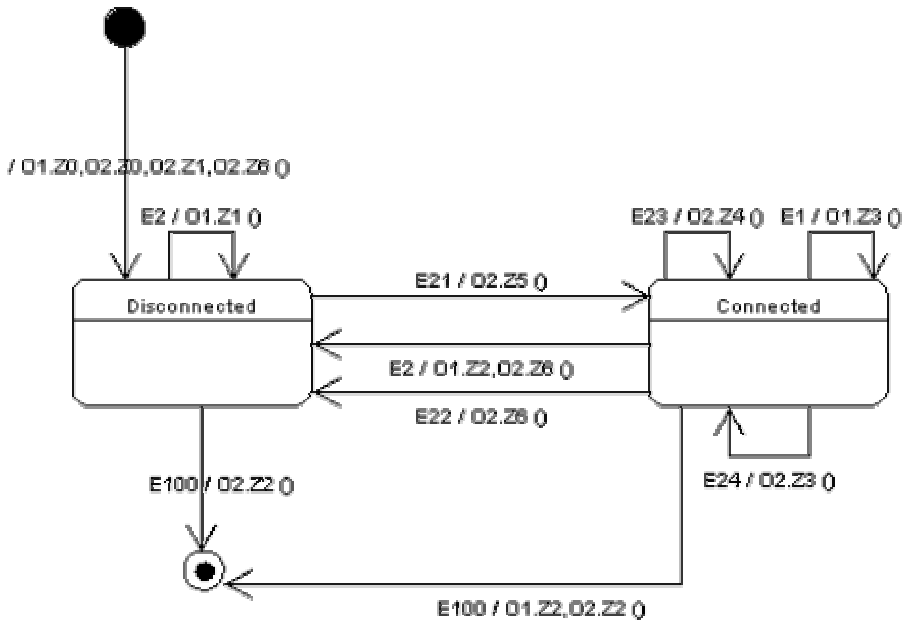

Рис. 7. Граф переходов клиентского автомата A2

С помощью разработанного конвертора [7] на основе диаграмм на рис. 6, 7 автоматически строится *XML*-описание автомата A2 (Листинг 2). Данное описание изоморфно указанным диаграммам.

Листинг 2. XML-описание клиентского автомата A2

```
<?xml version="1.0" encoding="UTF-8"?>
<stateMachine name="A2">

    <controlledObject name="O1">

<controlledObjectClassName>org.ifmo.messenger.client.co.ClientConnector</controlledObjectClassName>
    </controlledObject>

    <controlledObject name="O2">
        <controlledObjectClassName>org.ifmo.messenger.client.co.Screen</controlledObjectClassName>
    </controlledObject>

    <state name="TOP" type="0">
        <state name="Start" type="1"/>
        <state name="Connected" type="0"/>
        <state name="Disconnected" type="0"/>
        <state name="Final" type="2"/>
    </state>

    <transition>
        <sourceStateRef>Connected</sourceStateRef>
        <targetStateRef>Connected</targetStateRef>
        <event name="E23"/>
        <outcomeEffect>O2.Z4</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Disconnected</sourceStateRef>
        <targetStateRef>Connected</targetStateRef>
        <event name="E21"/>
        <outcomeEffect>O2.Z5</outcomeEffect>
    </transition>
```

```
    <transition>
        <sourceStateRef>Disconnected</sourceStateRef>
        <targetStateRef>Final</targetStateRef>
        <event name="E100"/>
        <outcomeEffect>O2.Z2</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Connected</sourceStateRef>
        <targetStateRef>Connected</targetStateRef>
        <event name="E1"/>
        <outcomeEffect>O1.Z3</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Connected</sourceStateRef>
        <targetStateRef>Connected</targetStateRef>
        <event name="E24"/>
        <outcomeEffect>O2.Z3</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Connected</sourceStateRef>
        <targetStateRef>Disconnected</targetStateRef>
        <event name="E22"/>
        <outcomeEffect>O2.Z6</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Disconnected</sourceStateRef>
        <targetStateRef>Disconnected</targetStateRef>
        <event name="E22"/>
        <outcomeEffect>O2.Z6</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Start</sourceStateRef>
        <targetStateRef>Disconnected</targetStateRef>
        <outcomeEffect>O1.Z0</outcomeEffect>
        <outcomeEffect>O2.Z0</outcomeEffect>
        <outcomeEffect>O2.Z1</outcomeEffect>
        <outcomeEffect>O2.Z6</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Connected</sourceStateRef>
        <targetStateRef>Disconnected</targetStateRef>
        <event name="E2"/>
        <outcomeEffect>O1.Z2</outcomeEffect>
        <outcomeEffect>O2.Z6</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Connected</sourceStateRef>
        <targetStateRef>Final</targetStateRef>
        <event name="E100"/>
        <outcomeEffect>O1.Z2</outcomeEffect>
        <outcomeEffect>O2.Z2</outcomeEffect>
    </transition>

    <transition>
        <sourceStateRef>Disconnected</sourceStateRef>
        <targetStateRef>Disconnected</targetStateRef>
        <event name="E2"/>
        <outcomeEffect>O1.Z1</outcomeEffect>
    </transition>
</stateMachine>
```

На рис. 8 показан скриншот одной из рабочих ситуаций разработанного приложения.

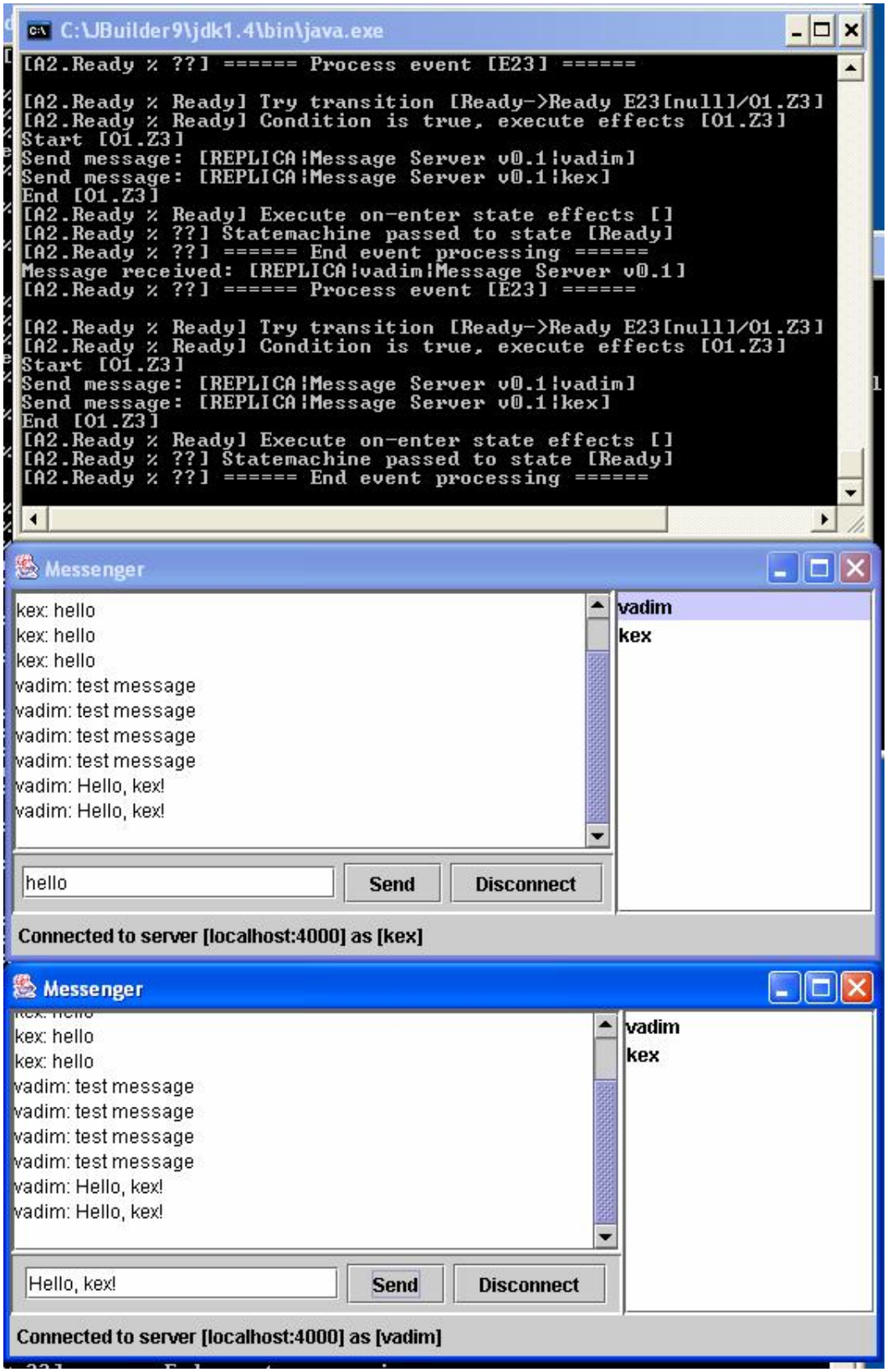

Рис. 8. Серверная консоль и два клиентских приложения

Верхнее окно – протокол работы серверного приложения, а нижние два – окна клиентских приложений. Клиентские приложения присоединены к серверу и обмениваются сообщениями. Из серверного протокола следует, что последним было обработано событие *E23* – получен текст. В протоколе также отображено, что при обработке этого события выполнено выходное воздействие *O1.Z3* – пересылка текста присоединенным клиентам. В нижних окнах приведены сообщения, которыми обменивались клиенты, пересылая сообщения друг другу через сервер.

6. Логирование

Приведем логи работы серверного и клиентского приложений. Для удобства лог имеет следующий формат:

[StateMachine.ActiveState % StateBeingProcessed] Message

В этом формате применяются следующие обозначения:

- *StateMachine* – название автомата;
- *ActiveState* – текущее состояние автомата (всегда простое);
- *StateBeingProcessed* – состояние, переход из которого в данный момент пытается сделать автомат. Это состояние может не совпадать с текущим, если из последнего выполнен переход в сложное состояние.

Лог для сервера имеет вид:

```
[A2.?? % ??] ====== Process event [null] ======

[A2.StartState1 % StartState1] Try transition [StartState1->Ready [null]/O1.Z0,O2.Z0]
[A2.StartState1 % StartState1] Condition is true, execute effects [O1.Z0,O2.Z0]
Start [O1.Z0]
Set listener [org.ifmo.messenger.server.Server@1971afc]
Start listener thread
End [O1.Z0]
Start [O2.Z0]
Set keyboard listener [org.ifmo.messenger.server.Server@1971afc]
Start keyboard thread
End [O2.Z0]
[A2.StartState1 % StartState1] Execute on-enter state effects []
[A2.Ready % ??] Statemachine passed to state [Ready]
[A2.Ready % ??] ====== End event processing ======
Start accepting connections on port 4000
New connection accepted [/127.0.0.1]
Create new thread for incomming connection
Message received: [LOGIN|vadim|Message Server v0.1]
[A2.Ready % ??] ====== Process event [E21] ======

[A2.Ready % Ready] Try transition [Ready->Ready E21[O3.X1]/O1.Z1,O1.Z5]
[A2.Ready % Ready] Condition is true, execute effects [O1.Z1,O1.Z5]
Start [O1.Z1]
Send login allowed message
Send message: [LOGIN|Message Server v0.1|vadim]
End [O1.Z1]
Start [O1.Z5]
Send user list to all connected users
Send user list to vadim
Send message: [USERS|Message Server v0.1|vadim]
End [O1.Z5]
[A2.Ready % Ready] Execute on-enter state effects []
[A2.Ready % ??] Statemachine passed to state [Ready]
[A2.Ready % ??] ====== End event processing ======
Message received: [REPLICA|vadim|Message Server v0.1]
[A2.Ready % ??] ====== Process event [E23] ======
```

```
[A2.Ready % Ready] Try transition [Ready->Ready E23[null]/O1.Z3]
[A2.Ready % Ready] Condition is true, execute effects [O1.Z3]
Start [O1.Z3]
Send message: [REPLICA|Message Server v0.1|vadim]
End [O1.Z3]
[A2.Ready % Ready] Execute on-enter state effects []
[A2.Ready % ??] Statemachine passed to state [Ready]
[A2.Ready % ??] ====== End event processing ======
Message received: [LOGOUT|vadim|Message Server v0.1]
[A2.Ready % ??] ====== Process event [E22] ======

[A2.Ready % Ready] Try transition [Ready->Ready E22[null]/O1.Z6,O1.Z5]
[A2.Ready % Ready] Condition is true, execute effects [O1.Z6,O1.Z5]
Start [O1.Z6]
End [O1.Z6]
Start [O1.Z5]
Send user list to all connected users
End [O1.Z5]
[A2.Ready % Ready] Execute on-enter state effects []
[A2.Ready % ??] Statemachine passed to state [Ready]
[A2.Ready % ??] ====== End event processing ======
Exception in client thread [Socket is closed]. Close connection and stop thread [/127.0.0.1]
Remove user [vadim] from internal map
[A2.Ready % ??] ====== Process event [E100] ======

[A2.Ready % Ready] Try transition [Ready->EndState1 E100[null]/O1.Z20,O2.Z20]
[A2.Ready % Ready] Condition is true, execute effects [O1.Z20,O2.Z20]
Start [O1.Z20]
Stop listener thread
Exception in main listener thread occured: socket closed
Stop thread.
End [O1.Z20]
Start [O2.Z20]
Stop keyboard thread
End [O2.Z20]
[A2.Ready % Ready] Execute on-enter state effects []
[A2.EndState1 % ??] Statemachine passed to state [EndState1]
[A2.EndState1 % ??] ====== End event processing ======
```

Лог для клиента имеет вид:

```
[A1.?? % ??] ====== Process event [null] ======

[A1.Start % Start] Try transition [Start->Disconnected [null]/O1.Z0,O2.Z0,O2.Z1,O2.Z6]
[A1.Start % Start] Condition is true, execute effects [O1.Z0,O2.Z0,O2.Z1,O2.Z6]
Start [O1.Z0]
End [O1.Z0]
Start [O2.Z0]
End [O2.Z0]
Start [O2.Z1]
End [O2.Z1]
Start [O2.Z6]
Clear user list
End [O2.Z6]
[A1.Start % Start] Execute on-enter state effects []
[A1.Disconnected % ??] Statemachine passed to state [Disconnected]
[A1.Disconnected % ??] ====== End event processing ======
[A1.Disconnected % ??] ====== Process event [E2] ======

[A1.Disconnected % Disconnected] Try transition [Disconnected->Disconnected E2[null]/O1.Z1]
[A1.Disconnected % Disconnected] Condition is true, execute effects [O1.Z1]
Start [O1.Z1]
Send message: [LOGIN|vadim|Message Server v0.1]
End [O1.Z1]
[A1.Disconnected % Disconnected] Execute on-enter state effects []
[A1.Disconnected % ??] Statemachine passed to state [Disconnected]
[A1.Disconnected % ??] ====== End event processing ======
Message received: [LOGIN|Message Server v0.1|vadim]
[A1.Disconnected % ??] ====== Process event [E21] ======

[A1.Disconnected % Disconnected] Try transition [Disconnected->Connected E21[null]/O2.Z5]
[A1.Disconnected % Disconnected] Condition is true, execute effects [O2.Z5]
```

```
Start [O2.Z5]
End [O2.Z5]
[A1.Disconnected % Disconnected] Execute on-enter state effects []
[A1.Connected % ??] Statemachine passed to state [Connected]
[A1.Connected % ??] ====== End event processing ======
Message received: [USERS|Message Server v0.1|vadim]
[A1.Connected % ??] ====== Process event [E24] ======

[A1.Connected % Connected] Try transition [Connected->Connected E24[null]/O2.Z3]
[A1.Connected % Connected] Condition is true, execute effects [O2.Z3]
Start [O2.Z3]
End [O2.Z3]
[A1.Connected % Connected] Execute on-enter state effects []
[A1.Connected % ??] Statemachine passed to state [Connected]
[A1.Connected % ??] ====== End event processing ======
[A1.Connected % ??] ====== Process event [E1] ======

[A1.Connected % Connected] Try transition [Connected->Connected E1[null]/O1.Z3]
[A1.Connected % Connected] Condition is true, execute effects [O1.Z3]
Start [O1.Z3]
Send message: [REPLICA|vadim|Message Server v0.1]
End [O1.Z3]
[A1.Connected % Connected] Execute on-enter state effects []
[A1.Connected % ??] Statemachine passed to state [Connected]
[A1.Connected % ??] ====== End event processing ======
Message received: [REPLICA|Message Server v0.1|vadim]
[A1.Connected % ??] ====== Process event [E23] ======

[A1.Connected % Connected] Try transition [Connected->Connected E23[null]/O2.Z4]
[A1.Connected % Connected] Condition is true, execute effects [O2.Z4]
Start [O2.Z4]
End [O2.Z4]
[A1.Connected % Connected] Execute on-enter state effects []
[A1.Connected % ??] Statemachine passed to state [Connected]
[A1.Connected % ??] ====== End event processing ======
[A1.Connected % ??] ====== Process event [E2] ======

[A1.Connected % Connected] Try transition [Connected->Disconnected E2[null]/O1.Z2,O2.Z6]
[A1.Connected % Connected] Condition is true, execute effects [O1.Z2,O2.Z6]
Start [O1.Z2]
Send message: [LOGOUT|vadim|Message Server v0.1]
End [O1.Z2]
Start [O2.Z6]
Clear user list
End [O2.Z6]
[A1.Connected % Connected] Execute on-enter state effects []
[A1.Disconnected % ??] Statemachine passed to state [Disconnected]
[A1.Disconnected % ??] ====== End event processing ======
Exception [socket closed]. Close connection and stop thread [Client Connector Listener]
[A1.Disconnected % ??] ====== Process event [E22] ======

[A1.Disconnected % Disconnected] Try transition [Disconnected->Disconnected E22[null]/O2.Z6]
[A1.Disconnected % Disconnected] Condition is true, execute effects [O2.Z6]
Start [O2.Z6]
Clear user list
End [O2.Z6]
[A1.Disconnected % Disconnected] Execute on-enter state effects []
[A1.Disconnected % ??] Statemachine passed to state [Disconnected]
[A1.Disconnected % ??] ====== End event processing ======
[A1.Disconnected % ??] ====== Process event [E100] ======

[A1.Disconnected % Disconnected] Try transition [Disconnected->Final E100[null]/O2.Z2]
[A1.Disconnected % Disconnected] Condition is true, execute effects [O2.Z2]
Start [O2.Z2]
End [O2.Z2]
[A1.Disconnected % Disconnected] Execute on-enter state effects []
[A1.Final % ??] Statemachine passed to state [Final]
[A1.Final % ??] ====== End event processing ======
```

7. Обеспечение возможности модификации системы

Рассмотрим ситуацию, когда необходимо внести изменения в логику функционирования системы. *Предположим, что весь код, реализующий выходные воздействия, уже написан и отлажен*. Предположим также, что появилась необходимость внести следующие изменения в систему:

- если клиент не подсоединен к серверу, то после нажатия кнопки *Send* сначала автоматически должно выполняться подсоединение к серверу, и лишь потом – отправка сообщения;
- нельзя закрыть клиентское окно, пока клиент подсоединен к серверу.

Проанализировав табл. 5, можно сделать вывод, что написанный код изменять не следует, а требуется изменить лишь граф переходов автомата A2. При этом в состояния *Disconnected* необходимо добавить петлю, помеченную символами *E1/O1.Z1,O1.Z3,* и убрать переход из состояния *Connected* в финальное состояние. На рис. 9 представлена модифицированная схема графа переходов клиентского автомата A2.

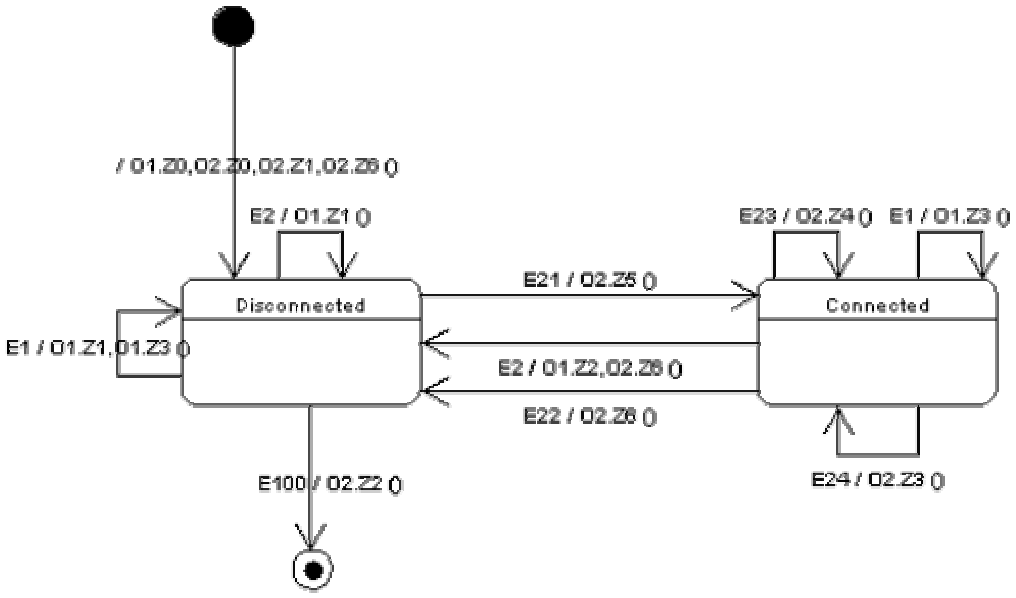

Рис. 9. Преобразованный граф переходов автомата A2

После этого достаточно сгенерировать новое XML-описание автомата A2, и система готова к работе.

Заключение

В работе приведен пример построения простого клиент-серверного приложения на основе автоматного подхода. При этом основой для разработки такого приложения является описание протокола взаимодействия между клиентами и сервером. Используя этот протокол, строятся графы переходов рассматриваемых приложений, а также описываются потоки данных.

Применяемый подход базируется на совместном использовании SWITCH-технологии и *UML*-диаграмм, которые создаются в свободно распространяемой среде разработки *ArgoUML* [3].

На основе анализа предметной области строится диаграмма классов системы, выделяются объекты и управляющие ими автоматы.

Схема связей каждого автомата изображается с помощью нотации диаграммы классов, а его граф переходов – в виде диаграммы состояний. Разработана программа – конвертер, преобразующая эти диаграммы в *XML*-описание, которое интерпретируется после «ручной реализации» функций входных переменных и выходных воздействий.

Правильность работы приложения демонстрируется с помощью логирования, выполняемого в автоматной терминологии.

Таким образом, если иметь широкую библиотеку функций, реализующих входные и выходные воздействия, то во многих случаях практически ничего не приходится программировать вручную. При этом создание приложения сводится к построению диаграмм, описывающих схемы связей автоматов и их графы переходов, и генерации по ним *XML*-описания логики работы системы. В дальнейшем полученное *XML*-описание, как отмечено выше, интерпретируется с помощью разработанной авторами среды выполнения.

Исходные тексты программы доступны по адресу
http://unimod.sourceforge.net/download.html.

В заключении отметим, что рассмотренный пример является весьма простым, однако у авторов есть основание предполагать [8], что он может быть распространен и на более сложные системы.

Литература

1. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. №2. **http://is.ifmo.ru**, раздел «Статьи».
2. *Гуров В.С., Шалыто А.А.* XML и автоматы. **http://is.ifmo.ru**, раздел «Проекты».
3. *ArgoUML*. **http://argouml.org**
4. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж*. Приемы объектно-ориентированного программирования. Паттерны проектирования. СПб.: Питер, 2001.
5. *Диаграммы потоков данных*. 1998. **http://lib.isystem.ru/programing/oop_rsis/glava~31.htm**
6. *Методология SA/SD*. 1999. **http://www.citforum.ru/programming/oop_rsis/glava4_2.shtml**
7. *Гуров В.С., Мазин М.А., Нарвский А.С.* UniMod. **http://unimod.sourceforge.net** .
8. *Гуров В.С.* Использование J2ME. **http://www.javable.com/columns/mobile/workshop/02/**

Приложение. Формальное описание правила формирования потока данных

1. Если на переходе задано логическое условие, то его следует записать в дизъюнктивной нормально форме.
2. Для каждой конъюнкции формируется последовательность вида $\{e_i, X_1, X_2, .., X_n\}$.
3. К каждой последовательности добавляется последовательность выходных воздействий на этом переходе $\{Z_1, Z_2, .., Z_m\}$.
4. К полученной последовательности добавляются все выходные воздействия $\{Z_{m+1}, .., Z_{m+k}\}$, записанные внутри состояния, в которое осуществляется переход.
5. Если переход осуществляется в сложное состояние, то необходимо выполнить шаги 1 – 4 для всех переходов, выполняемых из начального состояния сложного состояния. После этого необходимо добавить получившиеся последовательности к последовательностям, уже полученным после четвертого шага.
6. Шаги 1 – 5 выполняются для каждого перехода. Все последовательности записываются следующим образом: $\{P_1, P_2, .., P_{1+n+m+k}\}$.
7. Пусть B – множество событий, входных и выходных воздействий, а D – множество всех возможных данных системы. Введем функции:
   - $In(a) : a \subset B \rightarrow D$. Эта функция возвращает набор необходимых входных данных для любой входной переменной и любого выходного воздействия;
   - $Out(a) : a \subset B \rightarrow D$. Эта функция возвращает набор выходных данных для любого события, любой входной переменной и любого выходного воздействия;
8. Для всех полученных на пятом шаге последовательностей должно выполняться условие:

$$In(P_{i+1}) \subseteq \bigcup_{j=1}^{i} Out(P_j), \text{ где } i \subset [1, n+m+k].$$

Таким образом, объединение выходных данных всех элементов, предшествующих i-ому, должно покрывать множество входных данных i-го элемента последовательности.

Если правило формирование потока данных выполнено, то автомат корректно управляет потоком данных.