УДК 621.372, 681.142

ОСНОВНЫЕ ЭТАПЫ РАЗРАБОТКИ МУЛЬТИАГЕНТНЫХ СИСТЕМ В ИНСТРУМЕНТАЛЬНОЙ СРЕДЕ ДЛЯ СОЗДАНИЯ ИНТЕРНЕТ-ПРИЛОЖЕНИЙ

© 2003 С.В. Батищев[1], П.О. Скобелев[2]

[1] Самарский филиал Физического института РАН им. П.Н.Лебедева
[2] Институт проблем управления сложными системами РАН, г. Самара

Рассматриваются основные этапы разработки мультиагентного приложения сети Интернет. При этом особое внимание уделяется языковым средствам для представления и работы с онтологиями предметной области, конструирования архитектуры мультиагентного сообщества, описания конструкций и стратегий агента. На простом примере рассматриваются типичные приемы программирования мультиагентных систем в сети Интернет и анализируются основные особенности разработки таких систем в сравнении с традиционными средствами объектно-ориентированного программирования. Отмечается ряд особенностей проектирования серверных мультиагентных систем.

Введение

В условиях быстрого развития сети Интернет все более актуальной становится задача построения сложных сервисных и информационных систем. Программное обеспечение таких информационных систем должно отвечать ряду критериев: организация развитых online сервисов для пользователей, индивидуальный подход к пользователям ресурса, управление системой “на лету”, включая возможность легко уточнять описания предметной области, внедрять новые компоненты и сервисы.

В последнее время для решения таких задач предлагается мультиагентный подход. Такой подход позволяет описывать сложную систему как сеть небольших активных программных объектов – “агентов”. Агенты должны иметь способность использовать формализованное описание понятий и отношений предметной области – “онтологию” – в соответствии с некоторыми универсальными правилами и стратегиями поведения. Такой подход позволяет строить по настоящему открытые и самоорганизующиеся системы.

Основой создания таких систем может стать технология открытых мультиагентных систем (OMAC) корпорации MagentA, разрабатываемая в течение последних лет [1]. Так, например, для разработки мультиагентных приложений в сети Интернет можно удобно и эффективно использовать SB MagentA Engine (см., например, [2]).

Однако программирование мультиагентных Web-приложений имеет целый ряд особенностей:

- специфика мультиагентного подхода требует использования специальных языковых средств (например, управление параллельными процессами, управление онтологиями) для эффективного и выразительного описания;
- необходимо разрешить вопросы безопасности, устойчивости и расширяемости MAC для того, чтобы эффективно использовать ее в Интернет приложении на стороне сервера.

В данной статье мы рассмотрим основные этапы построения мультиагентного Интернет-приложения с помощью ядра SB MagentA Engine. При этом мы обратим особое внимание на языковые средства и визуальные инструменты, применяемые для реализации различных этапов мультиагентной системы.

Постановка задачи

Итак, продемонстрируем каким образом пользователи и программисты могут конструировать мультиагентное приложение с помощью ядра MagentA Engine и конструктора Ontology Toolkit. В качестве простого примера выберем простую систему мультиагентного планирования работ. Пусть в нашей системе будет только два типа агентов:

- агент программиста;

- агент проекта.

Задача этих агентов – провести переговоры и установить взаимовыгодное соответствие (“matching”) между проектом и программистом. В этом примере мы намеренно будем избегать использования уже готовых протоколов, стратегий агентов и других стандартных средств для того, чтобы рассмотреть все этапы разработки нового приложения. Однако при этом, мы упростим некоторые этапы разработки (оставив за кадром web-интерфейс, работы с базой данных, упростим структуру онтологии).

Создание проекта

После открытия MagentA Ontology Toolkit, необходимо создать новый проект, в котором мы будем разрабатывать все компоненты нашей системы. Проект – это коллекция файлов исходных текстов онтологии, шаблонов интерфейса, настроек интерпретатора ядра и т.п. Для создания проекта необходимо выбрать File → New Project и выбрать Simple Project. В этом случае, создастся пустой проект, в котором мы сможем “с нуля” начать конструирование. В проекте будет создана единственная сеть (main) в единственном файле.

Разработка протокола переговоров

С разработки протокола переговоров начинается первый этап разработки МАС – описание архитектуры мультиагентного сообщества.

На этом шаге необходимо разработать и сконструировать протокол переговоров агентов. Протокол описывает формальные правила, которым должны следовать участники переговорного процесса (использующие разные стратегии), чтобы их сообщения были понятны каждой стороне. В мультиагентном ядре описание протокола включает:

- имя протокола;
- имена сторон протокола (поддерживаются только двух сторонние протоколы, более сложные переговоры на данном этапе должны быть представлены как комбинация двухсторонних протоколов);
- сторона, которая начинает переговоры;
- типы, атрибуты и порядок сообщений, которыми могут обмениваться агенты.

Для создания протоколов переговоров имеется специальный мастер. Для его активации необходимо выбрать команду Insert → New Protocol (эта команда доступна только при открытом файле онтологии).

После того как мастер создаст протокол и его две стороны, онтолог должен визуально создать все сообщения и связать их отношениями следования (с помощь, команды Create Link контекстного меню).

В нашем случае ограничимся следующим протоколом:

- Агент заказа (проект) обращается с предложением (Query) к агенту ресурса (программист). Это сообщение имеет единственный атрибут – указатель на экземпляр онтологического класса, который предлагает для “матчинга” агент заказа (в нашем случае – указатель на объект “проект”).
- Агент ресурса отвечает согласием ResourceYes или отказом ResourceNo.

Конечно, в таком простом варианте протокол не сильно отличается по возможностям от традиционных средств поддержки полиморфизма в объектных языках (например, виртуальные функции). Но в более сложных ситуациях (циклические переговоры с взаимными подвижками, многовариантные переговоры), такая схема протоколов имеет существенные преимущества, явно публикуя доступную всем агентам системы схему переговоров.

Проектирование классов агентов

Для описания архитектуры мира мультиагентного сообщества необходимо определить

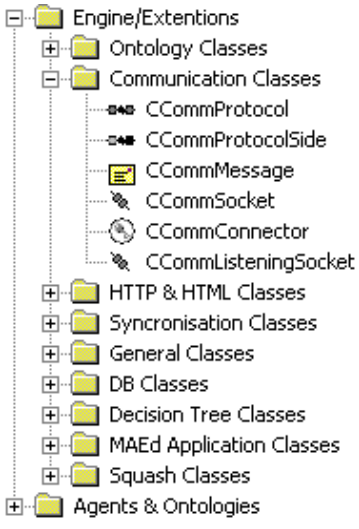

Рис.1. Список доступных классов

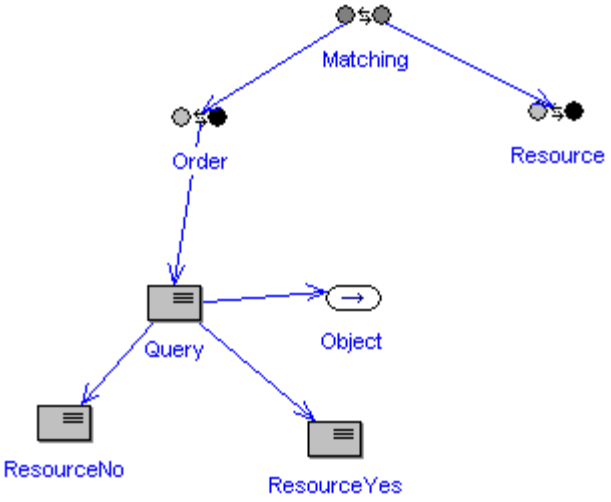

Рис.2. Протокол переговоров

классы агентов-участников. Агент в мультиагентном ядре – это объект специального типа CAgentClass (реализованный на C++), который имеет следующие специальные возможности и свойства:

- Имеет возможность владеть личными инструментальными объектами и стратегиями в своей сети. Специальным образом реализована поддержка объектов-соединителей (sockets), которые позволяют агентам инициировать переговоры.
- Имеет указатель на экземпляр онтологического объекта, которого он представляет. Разделение агента программиста и объекта "Программист" имеет важное методическое значение.
- Является потомком онтологического класса и поэтому сам может иметь атрибуты, связи и т.п.

Программисты и онтологи не обязаны создавать всех своих агентов на базе CAgentClass. При необходимости, программист может сам с помощью языка семантических сетей и сценариев сконструировать агентов из онтологических классов и других объектов. Однако управление такими агентами – более сложный процесс, который выходит за рамки этой статьи.

Агентов на основе CAgentClass можно создавать с помощью удобного мастера. Находясь на сети main, выберем команду File → New Agent.

Как видно на рисунке, при создании агента можно определить его основные параметры:

- имя;
- имя его личной (implementation) сети;
- имя онтологического класса, который представлен агентом (если необходимо);
- список протоколов, по которым агент может инициировать и принимать переговоры.

После завершения конструирования можно увидеть следующие изменения в сетях, указанные на рисунке:

Аналогично можно создать агента программиста с его онтологическим классом (не забыв указать, что он поддерживает сторону Resource в протоколе Matching, причем может принимать входящие запросы на переговоры).

В нашей системе агенты будут иметь всего по два строковых атрибута:

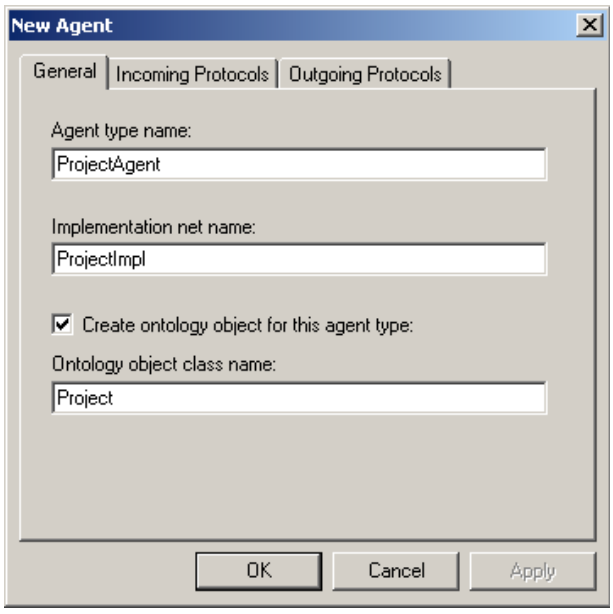

Рис.3. Создаем агента проекта с помощью мастера

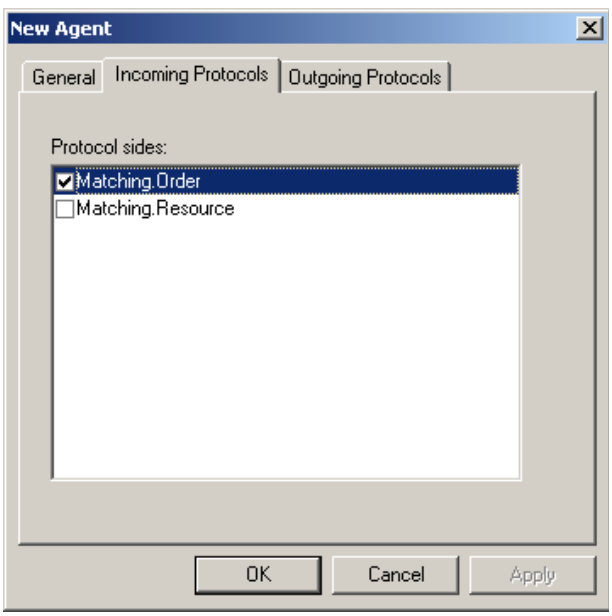

Рис.4. Определяем протоколы, по которым агент может переговариваться

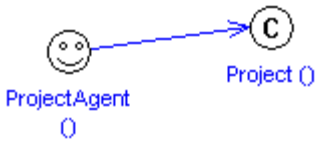

Рис.5. Агент и его онтологический класс

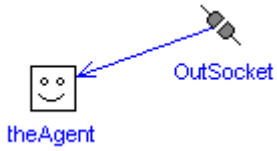

Рис.6. Личная сеть агента с указателем на самого агента и исходящим сокетом

- AgentName – в этом атрибуте будет указано имя агента для идентификации в логе и отладки.
- Condition – этот очень важный атрибут будет хранится условие (в виде сценария) принятия решения о матчинге. Значение этого атрибута будет интерпретироваться "на лету" стратегией агента.

После того, как классы агентов введены, необходимо определить *организацию* агентов. Простейшая организация агентов – неупорядоченный список агентов, который может наблюдаться любым агентом с соответствующей синхронизацией (COntHive).

Заполнение этого списка будем проводить при запуске системы с помощью специального автоматически стартующего инициализирующего сценария (Automatic Script).

Проектирование онтологии предметной области

Онтология – это концептуализация предметной области, доступная для работы агентам. В MagentA Engine онтология задается как семантическая сеть.

Мастер агентов уже создал классы Project и Programmer, которые будут являться основой нашей онтологии. Зададим теперь набор атрибутов этих классов. Пусть проект имеет:

- целочисленный атрибут Complexity (сложность проекта с минимальным значением 0 и максимальным 9);
- целочисленный атрибут Price (стоимость проекта).

А программист имеет:

- строковый атрибут Language (язык программирования, которым владеет).

Кроме того, пусть проект будет связан с программистом отношением "матчинга". В ядре MagentA Engine существует несколько способов представления отношений. В нашем случае наиболее просто воспользоваться однонаправленным ссылочным атрибутом (Link Attribute). Назовем этот атрибут Pair ("Пара").

На этом проектирование онтологии можно закончить и перейти к созданию сцены.

Конструирование сцены на основе онтологии предметной области и архитектуры мира

До того как мы перейдем к описанию конструкций агентов, создадим описание сцены (экстенсионального списка всех экземпляров агентов). Это даст нам возможность удобно отлаживать систему при конструировании агентов. Для этого дважды щелкнем по объекту InitScript сети main и внутри его сценария OnExecute опишем следующий сценарий:

```
var Agent1=ProjectAgent.Create();
pAgent1.AgentName=»Engine
    Development»;
pAgent1.Condition=»return
    Pair.Language==\»C++\» ||
    (Pair.Language==\»Java\»);»;
pAgent1.Object.Price=500;
pAgent1.Object.Complexity=3;
theHive.Add(pAgent1);
```

В этом сценарии мы выполняем:

- создание агента (автоматически создастся его онтологический объект);
- инициализацию атрибутов агента (AgentName и Condition);
- инициализацию атрибутов онтологического объекта агента (Price и Complexity);

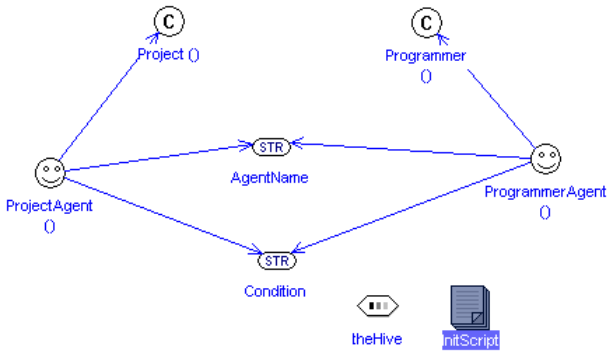

Рис.7. Агенты со всеми атрибутами, "ульем" и инициализирующим сценарием

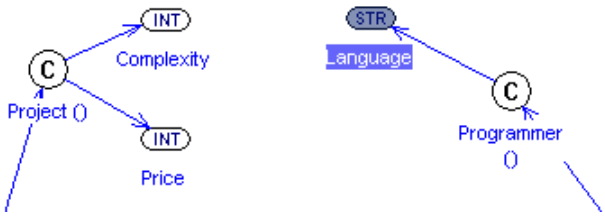

Рис.8. Атрибуты классов Project и Programmer

- добавляем агента в hive.

В реальных приложениях чтение сцены почти всегда осуществляется по результатам запроса к базе данных (а сам запрос зачастую формируется в результате анализа онтологии). Однако в нашем случае проинициализируем "вручную" второго агента:

```
var pAgent2=ProjectAgent.Create();
pAgent2.AgentName=»Engine
    Development»;
pAgent2.Condition=»return
    (Pair.Language==\»Java\»);»;
pAgent1.Object.Price=100;
pAgent1.Object.Complexity=1;
theHive.Add(pAgent1);
```

Совершенно аналогично можно произвести создание других агентов (агентов программистов):

```
var
    pAgent3=ProgrammerAgent.Create();
pAgent3.AgentName=»Jhon»;
pAgent3.Condition=»return
    Pair.Price>200 &&
    Pair.Complexity<3;»;
pAgent3.Object.Language=»C++»;
theHive.Add(pAgent3);

var
    pAgent4=ProgrammerAgent.Create();
pAgent4.AgentName=»Bob»;
pAgent4.Condition=»return
    Pair.Complexity<=1;»;
pAgent4.Object.Language=»Java»;
theHive.Add(pAgent4);

var
   pAgent5=ProgrammerAgent.Create();
pAgent5.AgentName=»Smith»;
pAgent5.Condition=»return
    Pair.Price>=400;»;
pAgent5.Object.Language=»Java»;
theHive.Add(pAgent5);
```

В конце инициализации необходимо выполнить для каждого созданного агента команду подтверждения создания. Эта команда синхронизации разрешает агентам начать исполнение сценарием. Если бы агенты начинали работать немедленно после создания (а работают агенты параллельно и автономно), то первые добавленные агенты увидели бы меньше агентов в списке, чем последние.

Обратите внимание, что атрибут Condition является обычным строковым атрибутом. А значит его значение тоже может хранится в базе данных и редактироваться пользователем. Таким образом пользователи сами могут задавать правила своему агенту. Так в нашем примере программисты задали следующие ограничения:

- Bob выполняет только самые простые проекты `Pair.Complexity`<=1.
- Smith берется только за дорогие проекты `Pair.Price`>=400.
- Jhon берется за не самые дешевые, но и не слишком сложные проекты `Pair.Price`>200 && `Pair.Complexity`<3

Безусловно, конечному пользователю удобнее пользоваться визуальным конструктором через Web-интерфейс, чтобы задавать правила своему агенту. Для этого предусмотрен специальный модуль, описание которого выходит за рамки этой статьи.

После выполнения этих действий, можно запустить проект на исполнение и убедиться в отсутствии синтаксических ошибок (Project→Run). Но для того, чтобы проект реально начал работать необходимо выполнить самый сложный этап – сконструировать устройство агента.

Разработка конструкции агента проекта

Начнем конструирование агента проекта с вывода агентов лог сообщений традиционной фразы "Hello, world". Перетащим в личную сеть агента Project (по умолчанию она называется ProjectImpl) объект автоматический сценарий. Назовем его MainScript и напишем там следующие команды:

```
object MainScript
{
  type=»CNodeAutoScript»;
  script OnExecute
  {
    dump.Trace(«Project: Hello
    World! I am the agent»+
```

```
        theAgent.AgentName);
      }
    }
```

Автоматический сценарий запустится после команды Commit (Вы можете убедиться в этом сами, расставив команды dump.Trace в тексте сценария InitScript, который мы создали в предыдущем разделе). После запуска проекта оба агента должны отрапортовать о своем создании в проколе работы ядра:

Service "NetLoader": initializing
Project: Hello World! I am the agent: Engine Development
Project: Hello World! I am the agent: Web Development
Service "NetLoader": initialization complete

Обратите внимание, что все действия и переговоры агентов, которые были созданы и возбуждены в автоматических сценариях, происходят в рамках инициализации ядра. Это очень важно для реальных веб-приложений – пока не закончится волна переговоров, возбужденных стартом системы, не будут приниматься запросы удаленных пользователей. Это решает многие проблемы синхронизации…

Описываем поиск партнеров для матчинга агента проекта

Первое, что должен сделать агент проекта – найти своих потенциальных "партнеров" для матчинга (агентов программистов).

Однако сценарии агентов лучше всего писать так, чтобы они минимально зависели от предметной онтологии. Поэтому не будем напрямую запрашивать агентов по именам, а обратимся к средствам навигации по онтологии.

Первое, что нам нужно сделать – это найти по экземпляру агента, который мы имеем в сети класс нашего агента:

```
var pMyClass=theAgent.Prototype;
```

Далее найдем класс объекта в онтологии, который мы представляем:

```
var pMyOntClass=pMyClass.ObjectClass;
```

После этого получим полный список атрибутов, которые имеет онтологический класс:

```
var pMyOntClassAttributes=
pMyOntClass.Attributes;
```

Теперь найдем в списке атрибутов атрибут, который имеет объектный тип:

```
var pMyLink=pMyOntClassAttributes.
Find(@A.TypeName=="object");
```

Обратите внимание, что в последней конструкции появился конструкт "@", который не имеет прямого аналога в C++ и Java. Это так называемое анонимное функциональное "замыкание" ("closure"). Подобные конструкции активно используются в смешанных (функционально-императивных) языках, например, lambda-функции в Python, замыкания в Vault (см. [3]). Когда условие предварили знаком "@", то оно не вычисляется, а передается как функция внутрь функции Find. Функция Find применяет выражение к каждому элементу последовательности и найдет только те элементы, которые удовлетворяют условию. Таким образом, мы найдем первый ссылочный атрибут.

И, наконец, получаем по указателю класс объект, который может стать нашей "парой":

```
var pMyPair=pMyLink.ObjectType;
```

Чтобы убедиться, что все прошло правильно, выполним отладочный dump.Trace:

```
dump.Trace("My pair class is"+
pMyPair.Name);
```

Убедившись, что все работает правильно, найдем теперь *всех* агентов в Hive, которые представляют интересующий нас класс (программисты) и "пробежимся" по их коллекции.

```
var MyPartners=main.theHive.
FindAll(@A.Prototype.ObjectClass==
pMyPair);
var nCount=MyPartners.Size;
for(var i=0;i<nCount;i++)
{
  var Partner=MyPartners[i];
  dump.Trace("My"+i+"potential
  partner is "+Partner.AgentName);
}
```

Теперь лог ядра должен включать список всех потенциальных партнеров.

Конечно, онтологию большого приложения нецелесообразно строить как точную копию этого простого примера. В реальном приложении было бы целесообразно выделить базовые классы ресурса и заказа, реализовав в них отношение "вступать в отношение матчинга с". Потом, в по-

томках этих классов можно было означить это отношение конкретными указателями на классы.

В этом случае, информацию о правилах соединения различных классов и их потомков знали бы не только сами агенты и программист, но и Ontology Toolkit, который может использовать эту информацию для работы с *библиотечными онтологиями*. Поскольку конструктор сам следит за правильностью конструирования как онтологий, так и сценариев, работа с такими онтологиями доступна менеджерам и экспертам.

На рис.9. показан экран при работе с онтологией мультиагентного интернет магазина.

Интерпретация агентом проекта сценария проверки условия на лету

Теперь агент должен принять решение: инициировать переговоры с потенциальным партнером или нет. В нашем случае агент использует для этого условие, заданное его пользователем:

```
var dynscript=
  Classes.CDynScriptContext();
dynscript.Pair=Partner.Object;
var Result=
  dynscript.Execute(theAgent.Condition);
if(Result)
{
  dump.Trace("I like him!");
}
else
{
  dump.Trace("I do not like him!");
}
```

Таким образом, стандартный класс CDynScriptContext позволяет задать набор именованных параметров (“Pair” – объект нашего найденного партнера-агента). В результате интерпретации и исполнения этого сценария “на лету” агент проверяет условие.

Инициирование и проведение переговоров со стороны агента проекта

Инициирование и проведение переговоров достаточно простой процесс, который предполагает использование следующих команд сокетов:

- *Соединить* – OutSocket.Connect(Partner). Операция возвращает коммуникационный сокет для обмена сообщениями.
- *Спросить* – sock.Ask(MessageClass, Attribute1, Attribute2). Операция посылает сообщение, ждет ответа и возвращает ответ.
- *Сказать* – sock.Ask(MessageClass, Attribute1, Attribute2). Операция посылает сообщение (но ответа не ждет).
- *Подождать сообщения* – sock.WaitForMsg(MessageClass). Операция ждет ответа.

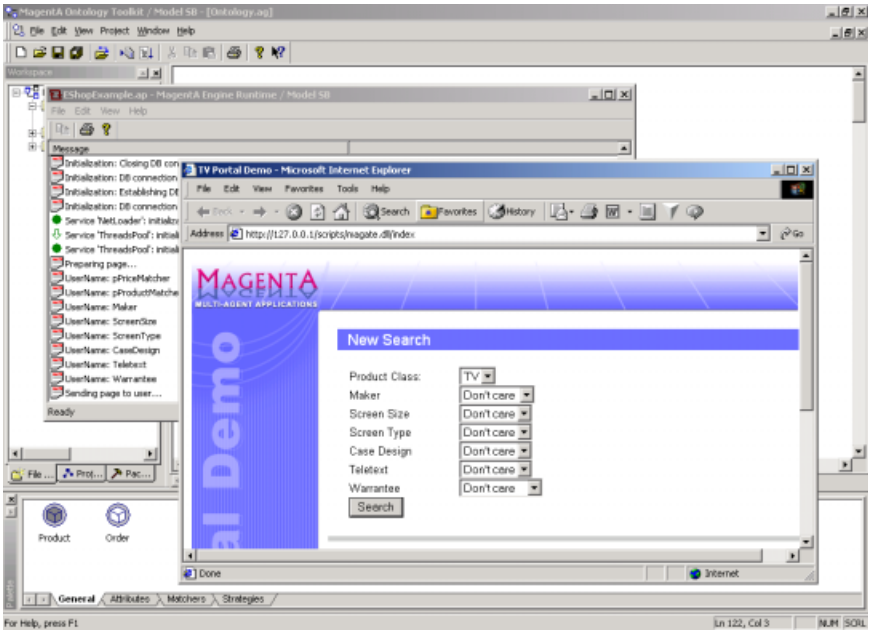

Рис.9. Запущенная система электронного магазина с “полкой” предопределенных базовых объектов на заднем плане

- *Отсоединиться* – sock.Disconnect(). Соединение разрывается.

Таким образом, в нашем случае сценарий переговоров будет выглядеть так:

```
if(Result)
{
dump.Trace("I like him!");
var sock=
    OutSocket.Connect(Partner);
if(sock)
{
var Reply=
    sock.Ask(Matching.Query,
    theAgent.Object);
if(Reply.Prototype==Matching.ResourceYes)
{
dump.Trace("My partner AGREED!");
dump.Trace("Link established:
  "+theAgent.AgentName+"
  "+Partner.AgentName);
}
else
{
dump.Trace("My partner DISAGREED.");
}
sock.Disconnect();
}
```

Если мы запустим приложение прямо сейчас и проанализируем лог, то убедимся, что произошла взаимная блокировка – агент послал сообщение, а ответа нет. В этом случае система исполнения отслеживает эту ситуацию и прерывает исполнение переговоров (если нет других) скриптов, которые могут продолжить работу.

Если бы переговоры проводились по запросу удаленного пользователя, то в этот момент очистились бы все ресурсы контеста памяти и закрылись транзакции СУБД. При этом запрос каждого удаленного пользователя обрабатывается в отдельном контексте и удаленный пользователь практически не имеет шансов "обрушить" систему. Однако такой подход накладывает целый ряд ограничений на стиль программирования. Самое главное из них – агенты из разных контекстов не могут видеть друг друга напрямую.

Конструирование агента программиста

Реализация пассивного агента программиста гораздо проще, чем реализация предыдущего агента. Единственное и главное отличие в том, что этому агенты не нужно запускать собственный автоматический скрипт, а достаточно ждать события прихода запроса на соединение (в интерактивном режиме). Это событие связано с объектом "слушающий сокет", который был создан в теле агента программиста мастером. Вставим в него реагирующий сценарий следующим образом.

```
object InSocket
{
type="CCommListeningSocket"»;
Agent = [object theAgent];
ProtocolSide =
  [object Matching.Resource];
script OnAccept
{
var sock=Trigger;
var pMsg=
  sock.WaitForMsg(Matching.Query);
var pObj=pMsg.Object;

var dynscript=
  Classes.CDynScriptContext();
dynscript.Pair=pObj;
var Result=
  dynscript.Execute(theAgent.Condition);
if(Result)
  sock.Say(Matching.ResourceYes);
else
  sock.Say(Matching.ResourceNo);
  sock.Disconnect();
}
```

Специальный идентификатор Trigger обозначает параметр (обычно "возбудитель") события. В случае события OnAccept таким "возбудителем" является коммуникационный сокет, через который можно осуществлять обмен сообщениями.

Заключение

Итак, мы полностью завершили простейшее мультиагентное приложение. Если мы запустим его на исполнение, то увидим, как агенты находят друг друга и устанавливают соединения в процессе переговоров.

На этом примере можно выделить некоторые ключевые особенности инструментальных средств ядра и стиля программирования агентов на базе ядра:

1. Знания предметной области можно формулировать независимо от реализации системы в виде онтологии предметной области.

2. Сценарии агентов можно писать независимо от предметной области с помощью опера-

ций навигации по онтологии.

3. Сценарии агентов могут включать динамические правила, загруженные из базы данных или сконструированные на лету.

4. Протоколы переговоров позволяют отдельно описать формальную схему процесса переговоров. Если два агента будут следовать этой схеме, то они смогут работать совместно, независимо от типа сценариев, которые ими управляют.

5. Для описания поведения агентов очень удобно использовать логику параллельных сценариев. Мультиагентное ядро предоставляет эффективный и прозрачный механизм диспетчеризации таких сценариев.

6. Функционально-императивное программирование позволяет удобно и наглядно описывать навигацию и изменение онтологии.

7. Ядро использует технологию контекстов (виртуальных "комнат") в рамках которых агенты производят переговоры. Если происходит ошибка, то все ресурсы контекста освобождаются и не влияют на работу ядра. Такой подход также позволяет очень эффективно управлять распределением памяти. Однако при этом нужно следить, чтобы агенты осуществляли переговоры отдельными независимыми блоками (в ответ на запрос пользователя или изменение в БД) длительностью не более 10 секунд. В этом случае ядро нагружается наиболее эффективным образом.

Отметим также некоторые правила, которые необходимо учитывать при разработке более сложных онтологий:

1. При соблюдении пунктов 1, 2 и 3 архитектурный уровень и фундаментальные метаконцепты онтологии можно отделить в отдельную библиотеку. В этом случае менеджер сможет управлять системой только на уровне предметной области и простого редактора правил агентов.

2. При создании приложений необходимо уделять внимание разделению отдельных слоев приложения. В нашем случае, были правильно выделены в отдельную сеть элементы конструкции агента. Но онтология и структура мира агентов находилась в одной сети, а это может быть неудобно для больших систем. Для конструирования многослойных сетей (с горизонтальными-внутрислойными и вертикальными-межслойными связями) удобно применять специальный "объект-проекцию" узла сети на другой слой).

СПИСОК ЛИТЕРАТУРЫ

1. *Batishev S., Ivkushkin K., Minakov I., Rzevski G., Skobelev P.* MagentA Multi-Agent Systems: Engines, Ontologies and Applications // Proceedings of the 3rd International Workshop on Computer Science and Information Technologies CSIT'2001. Ufa. Russia. 2001.
2. *Батищев С.В., Лахин О.И., Минаков И.В., Ржевский Г.А., Скобелев П.О.* Разработка инструментальной системы для создания мультиагентных приложений в сети Интернет // Известия Самарского научного центра РАН. 2001. №1.
3. *Mark Lutz, David Ascher.* Learning Python // O'Reilly, 1999.

MAIN STEPS OF MULTIAGENT SYSTEM DEVELOPMENT WITH WEB APPLICATION ENGINE

© 2003 S.V. Batishchev[1], P.O. Skobelev[2]

[1] Samara Branch of Physical Institute
named for P.N. Lebedev of Russian Academy of Sciences
[2] Institute for the Control of Complex Systems of Russian Academy of Sciences, Samara

Main steps of multiagent application development are proposed. Possible language tools for ontology representation, multiagent society definition and agent construction are analyzed. Sample application is proposed and multiagent system development process is demonstrated step by step for it. Particular features of multiagent application design are analyzed and compared with traditional object-oriented approach.