Софийски Университет „Св. Климент Охридски“
Факултет по математика и информатика
Катедра „Математическа логика и приложенията й“

Дипломна работа
на тема

Вграждане на COM обекти в Strawberry Prolog

на

Пламен Петков Шарапанов
Специалност „Информатика“, ф. № 41860

Научен ръководител:	Ръководител катедра:
Димитър Добрев	Проф. дмн. Иван Сосков

София 2006 г.

Съдържание

1 Въведение

Компонентния обектен модел (Component Object browser – COM) е неразделна част от съвременния Windows свят. Той е водещата „стандартна за индустрията“ софтуерна архитектура. При нея има програмно „разделение на труда“ между разработчиците на windows-базирани решения - създатели на използваеми модули и потребители, които ги вграждат в своите приложения. В края на миналия век от първия тип бяха C++ програмисти, запознати със заплетеността на COM теорията, а от втория VBA програмисти („Microsoft Excel – един по добър Visual Basic от Visual Basic“ [1]). В съвремието това разделение е заличено – възможно е да се напише COM компонент и COM клиент дори на VBScript. Съвсем естествено е желанието на пролог програмистите да могат да се възползват от тази утвърдена технология, а не да се принуждават да изучават и използват други езици.

Тази дипломна работа ще опише какви промени бяха направени в Strawberry prolog версия 2.6, за да имаме като резултат пролог програми използващи COM компоненти, както и външни приложения, ползващи пролог програми през COM.

Самата технология както и съпътстващата я терминология непрекъснато се променят (включително и в самия код на Microsoft). Основно ще използвам термините, добили популярност през годините – като водещи ще са понятията от C++ (тук както и на всякъде по-надолу под C++ имам предвид Microsoft Visual C++ версия 7.0). Те ще бъдат съгласувани с октомврийската версия на MSDN библиотеката [3], а съответните български означения ще са съгласувани с [7]. Понятията от пролог са спрямо версия 2.5 на Strawberry prolog [6].

1.1 Какво е COM

В тази глави ще дадем някои технически детайли относно COM света, чието познание е необходимо за самата реализацията на Strawberry prolog на C++, но които обикновенно пролог програмистите няма нужда да знаят, за да използват COM технологията.

Проблема с реализацията на междумодулна комуникация винаги е стоял пред програмистите и са били предлагани много различни решения. Стандартното решение за обмен на данни е да се ползва някакъв вид механизъм за обмен на съобщения – бил той през файлове, named pipes, sockets или пък специализиран – примерно Message Queue.

Комуникация от по-високо ниво може да се реализира примерно с използването на динамични библиотеки (DLL - Dynamic Link Library): клиентската програма зарежда по време на изпълнението си външен модул, за който има предварителна информация каква функционалност предлага и динамично я използва, а самата библиотека се грижи да целостта на данните си (примерно ползвайки споделена памет, ако е заредена от повече от един клиент). В Strawberry prolog има частична поддръжка на връзката с DLL библиотеки. Версия 2.5 може да извиква функции от външен DLL, но обратната посока не е реализирана (Strawberry prolog не може да бъде извикван като DLL библиотека).

Има технологии поддържащи обектно ориентирана междумодулна комуникация, които не са обвързани с Windows – примерно RMI (който е тясно интегриран с Java) и CORBA. CORBA е отворена платформено независима архитектура, позволяваща на различни приложения работещи на различни компютри с различни операционни системи да взаимодействат. Тя в много отношения е близка до COM и повечето от

принципите залегнали в тази дипломна работа, биха били приложими в една бъдеща нейна поддръжка.

COM предлага унифициран, разширяем, обектно ориентиран комуникационен протокол за Windows. Това е протокол, който свързва един софтуерен модул с друг, като комуникацията помежду им става посредством COM интерфейси - може да си ги мислим като непроменяем договор помежду им (като един обект може да реализира повече от един интерфейс). Комуникация е от тип господар–роб (master-slave). Господарят създава (или се прикачва към вече създаден) роб, изисква от него дадено действие (чрез интерфейсните му функции) и го освобождава. Господарят обикновено се нарича COM клиент, а робът – COM сървър. Ключово е, че COM е двоичен стандарт – комуникацията е единствено чрез интерфейси – няма входни точки, твърдо кодирани адреси, статични данни. Тази негова независимост от езика за разработка ни позволява да го реализираме в Strawberry prolog. Друго негово важно свойство е независимостта от местоположението (Location Transparency) - самите взаимодействащи модули могат да бъдат в едно и също адресно пространство, в различни процеси и дори на различни компютри (ползвайки DCOM – разпределен COM). Създаването на COM обекти става чрез извикването на системни функции. Всеки обект има собствен брояч, показващ броя на клиентите, които го реферират. Когато броячът стане нула, обектът се затваря автоматично.

1.2 Някои често срещани термини

Езикът за описването на интерфейсите е IDL (Interface Definition Language), като синтаксисът му е подобен на C.

В зависимост от това дали сървърът се намира на същия компютър, където е клиентът или на отдалечен, се нарича съответно локален сървър (local server) или отдалечен сървър (remote server). В зависимост от това дали сървърът работи в пространството на клиента или не, се нарича – вътрешно процесен сървър (In-process server) или външно процесен сървър (Out-of-process server). Обикновено, когато сървърът се реализира като DLL е вътрешно процесен сървър, а EXE е външно процесен сървър.

Всеки сървър се идентифицира със своя CLSID (Class identifier, идентификатор на класа) – уникално за вселената 16 байтово число или със своя ProgID (programmatic identifier, идентификатор на програма) – текстов низ от вида:

`<библиотека>.<компонента>.<версия>.`

и обикновено се съхраняват в регистъра на Windows.

Любопитни факти от историята:

В началото (около 1991 година) се е използвал терминът OLE (Object Linking and Embedding, свързване и влагане на обекти). Задачата му е била създаването на съставни документи, позволяващи примерно таблици от EXCEL да бъдат слагани директно в WORD документ. През 1993 г., с излизането на версия 2.0, под OLE вече се разбира технология за работа с компонентни обекти. Една от новите възможности (която няма нищо общо със свързването и влагането) е Automation, позволяваща на компонент и приложение да взаимодействат независимо от средствата и езика за програмиране. През 1996 г. при сблъсъка на „старото“ OLE с Интернет се

въвежда в обръщение терминът ActiveX, който има горната дефиниция, а OLE отново се ползва само за съставни документи.

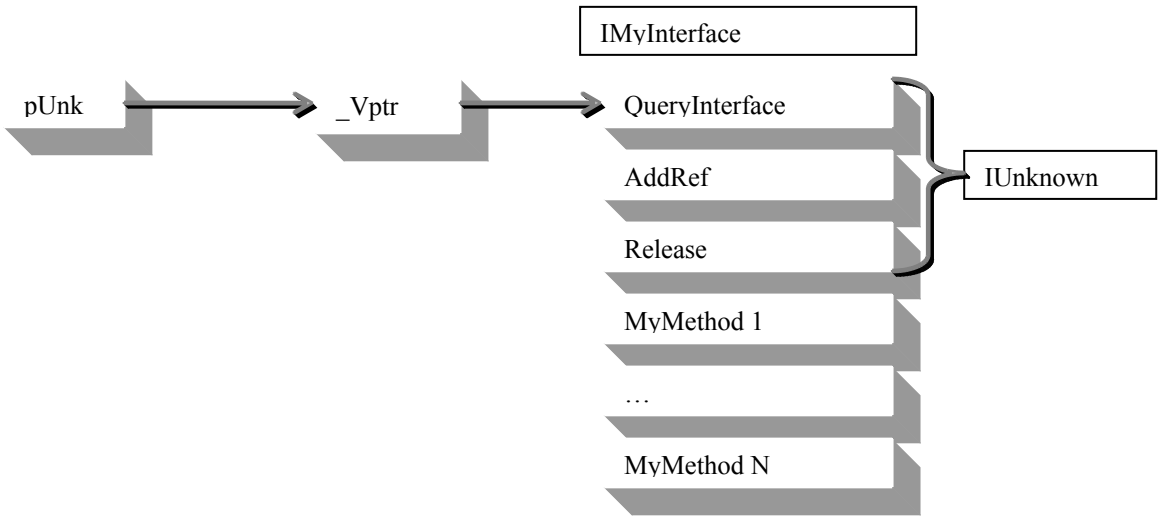
В началото езикът за описване на типовете е бил ODL (Obejct Definition Language), в последствие е заменен с IDL, който пък е базиран на спецификацията DCE RPC (distributed computing environment, Remote Procedure Call - отдалечено викане на методи в разпределени среди за изчисление) с добавена поддръжка на COM интерфейси.

В последно време в литературата COM сървър се среща като COM компонент, за да не се бърка със сървърския клас софтуер на Microsoft – примерно MS SQL Server.

Всъщност CLSID представлява GUID (globally unique identifier, глобално уникален идентификатор), което пък е реализацията на Microsoft на UUID (universally unique identifier, вселенски универсален идентификатор).

1.3 IUnknown

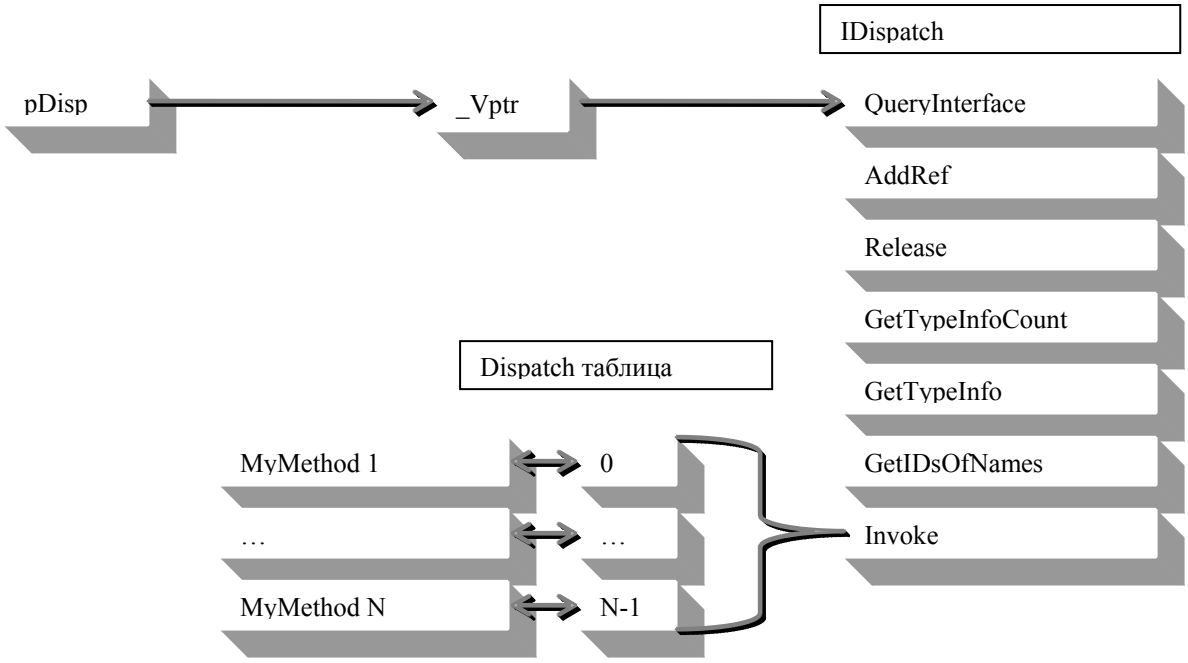
Всеки COM интерфейс и COM компонент трябва да реализира стандартния интерфейс **IUnknown**:



На Windows ниво COM обект се създава обикновено чрез извикване на API функцията CoCreateInstanse, която създава обект и връща указател към него от тип IUnknown, чийто вътрешен брояч е със стойност 1. Ако подадем този указател на друг клиент, той трябва да увеличи брояча му, извиквайки метода **AddRef**. Когато обектът повече не е необходим, се извиква неговия метод **Release**, който намалява брояча с единица и автоматично проверява дали е станал нула (т.е. дали това е бил последният активен интерфейс към обекта). Ако това е така, то самият обект се самоунищожава. Когато клиентът иска да работи с друг интерфейс на обекта, той извиква метода му **QueryInterface** с аргумент идентификатора на искания интерфейс. Ако обектът може да му го предостави му връща указател и увеличава брояча си с единица. Трябва да подчертаем, че COM клиентът никога не получава указател към истинския обект, а винаги указател към интерфейс, който е наследник на **IUnknown**.

1.4 IDispatch и двойнствени интерфейси

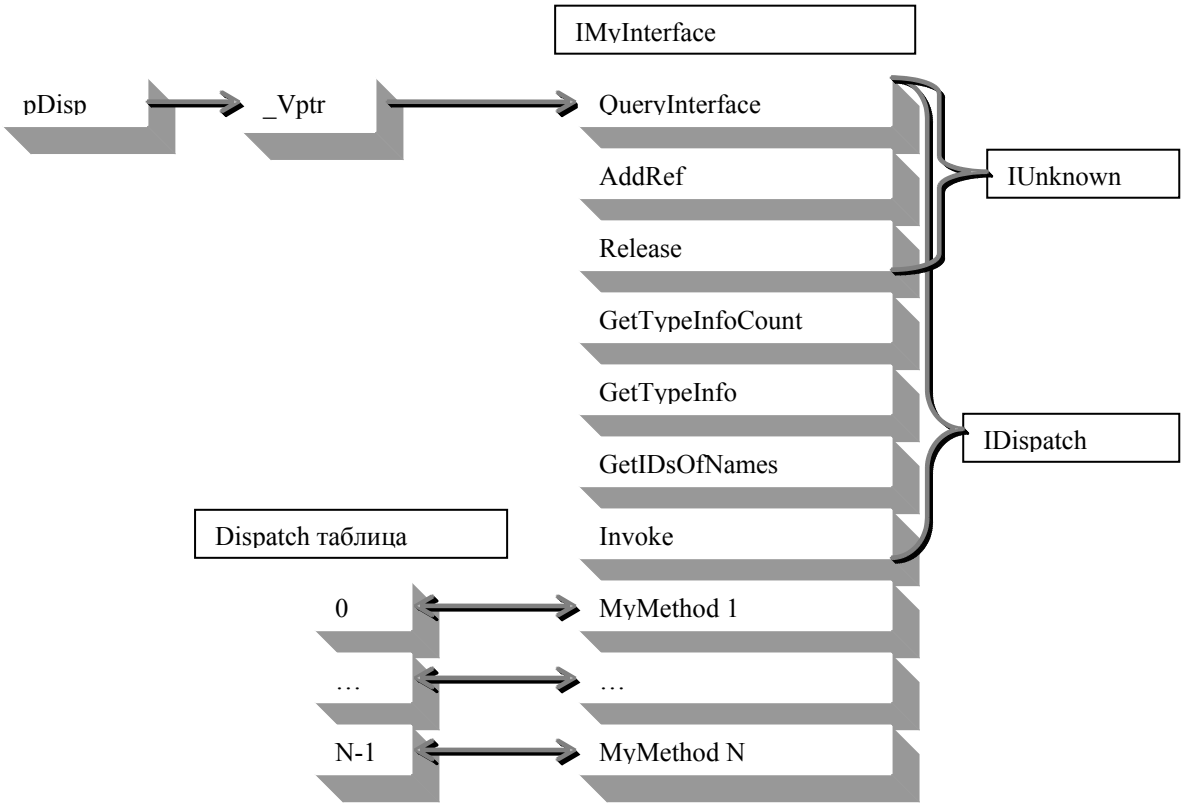
При използването на множество специализирани интерфейси проблемът е, че трябва да се измислят (и съответно да не се променят) и реализират и в клиента и в сървъра, което не винаги е удачно, примерно при скриптови клиенти. Решението е да се ползва многоцелеви интерфейс, който да „прекарва като през фуния“ цялата междумодулна комуникация [1]. Тази технология се нарича автоматизация (automation), а самият интерфейс е **IDispatch**:



Основният метод е **Invoke**, който има 8 аргумента, които определят кой метод на компонента ще бъде извикан, аргументите, които ще му бъдат подадени и като резултат връща резултата от извикания метод. Информация за типовете, които предоставя обектът може да се придобие чрез извикването на **GetTypeInfoCount** последвано от извиквания на **GetTypeInfo**. От гледана точка на VBA програмистите свойствата и методите имат символни имена, докато C++ програмистите използват по-

ефективните целочислени индекси. Ползвайки метода **GetIDsOfNames** може да се вземе индексът на метода по неговото име.

Ако свойствата и методите са достъпни само през IDispatch, то интерфейсът се нарича **dispinterface**. Обикновено чист dispinterface се използва, когато динамично (по време на изпълнението на програмата) се налага добавянето на нова функционалност. Но по-често срещаната ситуация е, когато сървърът предоставя едновременно и двата начина за достъп до функционалността си – през един естествен интерфейс и една Dispatch таблица. Такива интерфейси се наричат двойнствени (**Dual interface**).



2 Strawberry prolog като COM клиент

Както се видя от предишната глава, управлението на COM сървър не е тривиална задача – клиентът трябва да може да работи поне с два интерфейса, а и всяко извикване на метод си е приключение (споменете си 8-те аргумента на IDispatch.Invoke). Целта ни бе така да реализираме поддръжката на COM, че на пролог програмиста да не му се налага да е запознат с всички тези технически детайли. До версия 2.5 реализацията на тази поддръжка бе в доста лошо състояние – примерно не можеше да се извика метод с повече от един аргумент. В настоящата версия 2.6 постигнахме стабилност в работата с COM компонентите и въпреки добавената нова функционалност запазихме лекотата, с която се използват. От пролог програмиста не се изисква предварително познание за COM света. Достатъчна е базовата престава за обект като програмна структура, която съдържа в себе си данни и функционалност, които са достъпни единствено през публичните й интерфейси.

В тази глава ще опиша как се създават и унищожават COM обекти, където моят принос е основно в стабилизацията на съществуващия код; как се работи с методи и свойства - освен стабилизацията направих промени по обработката на някои типовете данни; и накрая реализирах две новите функционалности – поддръжка на изходящи интерфейси (обработка на събития) и вграждане на визуални COM компоненти.

2.1 Жизнен цикъл на COM обект

В Strawberry prolog създаването на екземпляр на COM обект става с вградената функция **server**:

```
X is server("COM_ID")
```

Горния ред се интерпретира от ядрото така:

1. Създай инстанция на COM обекта по COM_ID (където COM_ID е ProgID или CLSID) и вземи неговия dispinterface.
2. Създай пролог обект от тип сървър и запази в него този dispinterface.
3. Запази пролог обекта в глобалния склад на създадените обекти.
4. Свържи променливата X с този пролог обект.

Когато искаме да цитираме вече създаден обект през нова променлива:

```
Y = X
```

Зад кулисите Y всъщност се свързва към стойноста на X, а не се взима нов интерфейс към обета (и следователно не се увеличава броячът на COM обекта). Тази конструкция не е типична за COM програмирането и обикновено е предпоставка за грешка – тъй като клиентът може по невнимание да извика два пъти Release, но е напълно в духа на пролог. Може да си го мислим като конструкция от вида **const reference**, като синонимите (alias) са „обещали“ да не викат Release.

Обектът може да умре по два начина – или чрез извикване на вградения предикат **close**, или при автоматичното почистване на паметта, извършвано от ядрото на Strawberry prolog.

Използването на вградения предикат **close:**

```
close(X)
```

се интерпретира като:

1. Вземи обекта X.
2. Вземи съответния dispinterface на този обект.
3. Освободи инстанцията.

Трябва да се отбележи, че въпреки че обектът няма да има активни интерфейси от страна на пролог, може и да не умре, тъй като броячът му може да не е нула. Също така всички пролог променливи, сочещи към този обект ще си запазят стойността, но тя ще сочи към обект, маркиран като мъртъв в глобалния склад.

При почистването на паметта също се вика вътрешната реализация на **close**, но в този момент няма активна програма, така че няма да могат да се наблюдават страничните ефекти.

Може би пролог пуританите биха възнегодували срещу първия вариант, тъй като се получава силен страничен ефект (един вид насилствено развързване на вече свързана променлива), докато COM пуританите не биха възприели втория - COM-а е детерминистичен по самата си дефиниция и да се остави на „случайността“ да реши кога ще се извика Release методът е странно. Предложеното решение е подобно на Dispose идиома от C# - ако потребителят знае кога иска да си освобождава ресурсите, нека да му позволим да го прави, а ако не знае – ядрото ще го направи, когато реши, че обекта може да бъде освободен без това да създава проблеми.

Пример:

Следната примерна програма не реализира собствена логика за освобождаването на COM обектите си. След стартирането й и преди натискането на клавиша за затваряне, в управление на задачите ще видите 10 екземпляра на Strawberry.

```
?-
    test.

test :-
    for(I, 1, 9),
        X is server("Strawberry.Server"),
    fail.

test :- message("","press any key when ready",!).
```

Следващата програма сама освобождава COM обектите си. След стартирането й и преди натискането на клавиша за затваряне, в управление на задачите ще видите само един екземпляр на Strawberry.exe.

```
?-
    test.

test :-
    for(I, 1, 9),
        X is server("Strawberry.Server"),
        close(X),
    fail.

test :-message("","press any key when ready",!).
```

Последната програма е пример за COM сървър, който не умира при close:

```
?-
     X is server("Word.Application"),
     close(X).
```

Вижда се в управление на задачите, че дори след завършване на изпълнението на програмата и затварянето на средата, WINWORD.EXE остава в списъка на активните задачи. За да се затвори WINWORD.EXE трябва да се извика изрично командата й Quit:

```
?-
     X is server("Word.Application"),
     X.'Quit'(),
     close(X).
```

Комуникация от клиента към сървъра

Извикването на метод на COM обект от Strawberry prolog става по следния начин:

```
Res is X.'MethodName'(Arg1, … ,ArgN)
```

Където:

- MethodName е името на метода, който ще се извиква.
- Arg1, … ,ArgN са аргументите на извиквания метод.
- В Res се записва резултата от изпълнението на метода.

Променливата за резултат не е задължителна, тоест метода може да се извика като предикат вместо като функция. Списъка на аргументите може да има произволна дължина, като може да бъде дори и празния списък. Примерно:

```
X.'MethodName'()
```

Прочитането на стойността на свойство става с конструкция от вида:

```
Res is X.'PropName'
```

а попълването му:

```
X.'PropName' := Arg
```

Strawberry prolog реализира горните конструкции като първо по символното име намира съответния DISPID, после прехвърля стойностите на аргументите от пролог типове към COM типове, извика подходящия Invoke метод на COleDispatchDriver (който вътрешно извиква искания метод на обекта), взема резултата от извикването, преобразува го към пролог тип и попълва променливата за резултата.

Пример:

Класическия "Hello, world!" се реализира със следната програма:

```
?-
    WordApp is server("Word.Application"),
    WordApp.'Visible':=true,
    WordApp.'Caption':=
        "Strawberry Prolog feeds MS WORD",

    ColDoc is WordApp.'Documents',
    ODoc is ColDoc.'Add'(),

    ColPara is ODoc.'Paragraphs',
    OPara is ColPara.'Add'(),
    ORange is OPara.'Range',
    ORange .'InsertBefore'("Hello, world!"),

    message("","pess any key when ready",!),
    ODoc.'Close'(0),
    WordApp.'Quit'().
```

Като резултат се стартира приложението WORD, прави се видимо и се попълва лентата на заглавието му. След това се добавя нов документ в колекцията му от документи. В него се добавя параграф, в който се добавя област и в нея се изписва "Hello, world!". Програмата чака да се натисне клавиш и затваря приложението WORD.

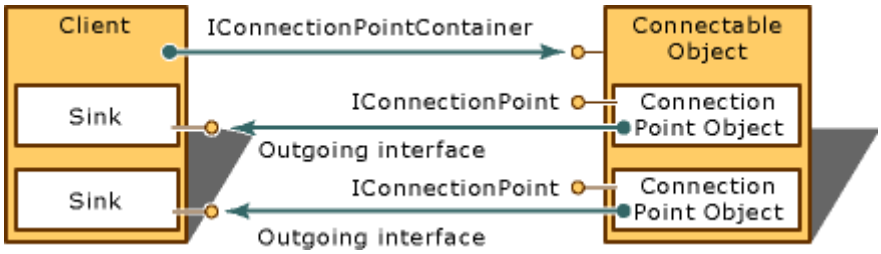
Подробности относно обектния модел на WORD, както и на другите приложения от офис пакета на Microsoft могат да се намерят в [3].

2.2 Комуникация от сървъра към клиента

Освен комуникацията от клиента към сървъра, имаме нужда и от обратна връзка, която да се осъществява от сървъра към клиента. Тази възможност се появява във версия 2.6 и нейното разработване е съществена част от тази дипломна работа. Тук ще разгледаме въпроса как се осъществява тази връзка, а накрая ще покажем какво е нужно на пролог програмиста и колко лесно той може да използва тази възможност.

Основно такъв вид комуникация е необходима, когато сървърът иска да уведоми клиента за промяна в състоянието си, но може да има ситуации, в които клиентът да не е готов да посрещне такова съобщение. Стандартното решение на този проблем е комуникацията винаги да се инициира от клиента – примерно повечето Windows драйвери през определен, достатъчно малък период от време отправят запитване за статуса на повереното им устройство и в зависимост от резултата реагират по съответен начин. Но и тази схема си има недостатъци – не работи в реално време, т.е. има забавяне от настъпването на събитието в сървъра до момента, в който ще бъде обработено в клиента – ако събитията възникват доволно често, то сървърът би трябвало да поддържа опашка за всеки клиент със събития, чакащи да бъдат прочетени от него, което би довело до усложняването на логиката на сървъра и занимаването му с несвойствени задачи.

В COM проблема се решава по следната схема:



1. Сървърът имплементира една или повече точки за свързване (connection point) на слушатели и дефинира изходящ интерфейс (Outgoign interface) за всяка точка.
2. Клиентът взима IConnectionPointContainer на сървъра.
3. Намира си от IConnectionPointContainer търсената IConnectionPoint, за която иска да бъде уведомяван (търсенето става по REFIID – идентификатор на изходящия интерфейс, което разбира се е GUID).
4. Клиентът създава собствен COM обект (наричан event sink) за искания тип събития, който реализира искания от сървъра интерфейс.
5. Подава го на сървъра, за да се абонира като слушател.
6. Сървърът го добавя в колекцията си от слушатели за типа събитие (преобразувайки аргумент 1 на Advise метода от IUnknown към изходящият интерфейс).
7. Когато настъпи събитие от този тип сървърът вика съответния метод на всички регистрирани слушатели.
8. Слушателя изпълнява извикания метод.

За нас са интересни само изходящите dispinterface, така че нашият слушател трябва да имплементира само IDispatch и то само метода му Invoke, който ще бъде викан от сървъра. Има много статии как да се реализира слушател, но всички са за ранно свързване, докато на нас ни трябва късно. Проблемът е, че предварително не знаем GUID-а на интерфейса, който би трябвало да имплементираме. В [4] е описан подобен проблем за Visual Basic клиент. Предложеното там решение е да се промени COM_MAP, така че всяко искане към QueryInterface за наследник на IDispatch да успява и всеки Invoke да се прихваща и препраща към клиентската програма. Използвайки го за база с няколко модификации и добавяйки механизъм за уведомяване в пролог, успяхме да го реализираме в настоящата версия.

За пролог програмиста всичко това е скрито и реално не му се налага да го знае – достатъчно му е да се абонира като слушател за събитията, идващи от сървъра и да обработи само тези, които са му необходими, ползвайки следната схема:

```
?- X is server2("COM_ID", on_event).
...
on_event::M(X) :- write(X).
```

В резултат се създава обект COM_ID, абонираме се за всички негови събитията и при настъпване на събитие се изпълнява предиката on_event с аргумент – името на събитието и списъка от аргументите му.

Пример:

Следната програма показва как може да се използва Agent технологията на Microsoft и как могат да се прихванат събитията, идващи от Мерлин (пробвайте да го преместите или да натиснете с бутона на мишката върху него):

```
?-
     StrAgentName is "Merlin",
     StrAgentPath is
          windows_path
          + "\\Msagent\\Chars\\"
          + StrAgentName + ".acs",
     ObjAgent is
          server2("Agent.Control.2",on_event),
     ObjAgent.'Connected' := 1,
     ColCharacters is ObjAgent.'Characters',
     ColCharacters.'Load'(
          StrAgentName, StrAgentPath),
     ObjCharacter is
          ColCharacters.'Character'(
               StrAgentName),

     ObjCharacter.'Show'(),
     message("","pess any key when ready",!),
     ObjCharacter.'Hide'(),
     repeat,
     wait(0.3),
     IsVisible := ObjCharacter.'Visible',
     IsVisible = false,
     close(ObjAgent).



on_event::M( | L):-
     write("on event "),
     write(M( | L)), nl,
     G_Return:=0.
```

Разбира се в on_event вместо пасивно записване на събитията може да се предприемат различни действия в зависимост от събитието. Примерно, ако направим ObjCharacter глобална променлива G_ ObjCharacter в горната програма и променим логика при обработката на събитие:

```
on_event::M(|L):-
    process(M).

process('DblClick') :-
    G_ObjCharacter.'Play'("Pleased").
```

То Мерлин ще се усмихва при всяко двойно щракване на мишката върху него.

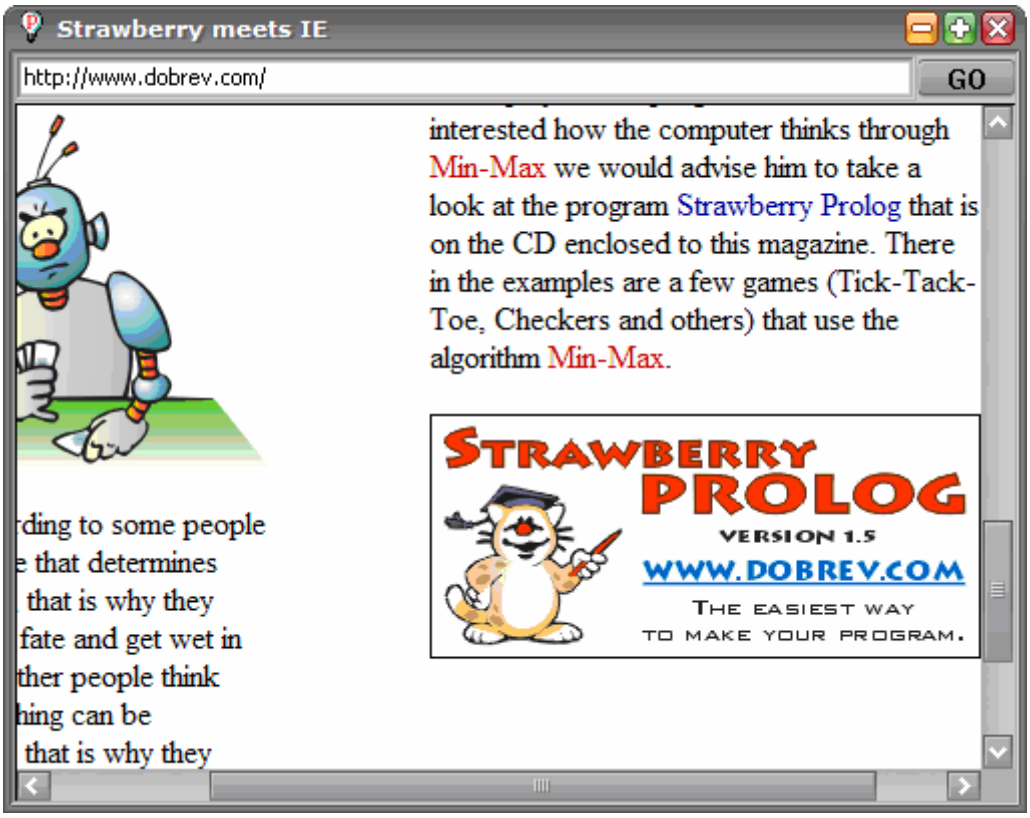
2.3 Работа с оле сървъри

Ако се върнем към корените на COM една от основните му цели е била готови визуални компоненти да могат да бъдат вмъквани и използвани като стандартни контроли в клиентските приложения. За клиента трябва да е еднакво лесно както да вмъкне дъщерен прозорец (примерно бутон) в основния си прозорец, така и визуална компонента [5].

Това бе реализирано с помощта на ATL библиотеката – ползвайки класа й CAxWindow. Класът предлага междинен прозорец, който се вмъква в клиентското приложение и който реализира интерфейсите, необходими на визуалната компонента да се вмъкнат в него.

В Strawberry prolog бе добавен предиката **embedserver**, който има освен всички аргументи на window и един допълнителен - задаващ сървъра, който ще се вложи в него. Сървърът може да е бъде създаден предварително или да се зададе по CLSID или ProgID.

Следната програма демонстрира колко лесно може да си направим браузър:




```
?-
  window( G_W, _, wnd_func(_),
    "Strawberry meets IE",   10, 100, 511, 406).

wnd_func(init) :-
  STR_URL := "http://dobrev.com/",
  G_Obj is server("Shell.Explorer.2"),
  embed_server( G_AX_IE, _, _, "",
    0, 24, 500,350, G_Obj),
  edit( G_EDIT_URL, _, edit_func(_), STR_URL,
    0, 0, 450, 23),
  button( G_BTN_GO, _, but_func_go(_), "GO",
    450, 2, 50, 20),
  G_Obj.'Navigate'(STR_URL,0,0,0,0).

but_func_go(press) :-
    STR_URL is get_text(G_EDIT_URL),
    G_Obj.'Navigate'(STR_URL,0,0,0,0).
```

Вижда се, че G_AX_IE и примерно G_EDIT_URL се използва по един и същи начин – независимо от факта, че първото е COM обекта за Internet Explorer, а второто стандартно текстово поле.

3 Strawberry prolog като COM сървър

Strawberry prolog е реализиран като COM сървър и съответно може да извиква сам себе си през COM. До версия 2.5 самите документи бяха реализирани като сървъри. В резултат при първото искане за нов обект се стартира приложението и се отваря нов документ в него, а при следващите създавания се използва същия процес и само се отварят нови документи (всъщност не е точно така – описаната схема важи само, когато потребителят е един и същ, в противен случай се правят проверки на правата, но тази ситуация няма да бъде разглеждана). И тук възниква проблем – архитектурата на ядрото на Strawberry prolog, което изпълнява самите пролог програми, е една инстанция за процес (SingleInstancePerProcess). Докато горният подход предполага обратното – много конкурентни документи всеки изпълняващ по една пролог програма. Идеята за пренаписване на ядрото, така че да може имаме по един негов екземпляр във всеки документ бе крайно нереалистична и затова трябваше да се потърси друго решение, а самата задача се преформулира по следния начин:

1. Да може да се създават COM обекти в процеса.
2. Най-много един COM обект да може да бъде създаден в процеса.
3. Ако процесът е стартиран от потребителя да не се ползва за COM.
4. След последното освобождаване на създадения COM обект, процесът трябва да се затвори самичък, без да се викат допълнителни методи (а не както случая с WORD в т. 3.1.1).
5. Обектът да може да уведомява клиентите си за събития.
6. В един обект да има най-много една пролог програма.

Като резултат във версия 2.6 реализирах един нов външно процесен COM сървър с двойнствен интерфейс за управление и един изходящ интерфейс за събития. Клиентското приложение единствено трябва да спазва базовия сценарий за управление, а от пролог програмиста отново не се изискат предварителни познания за COM, за да може да го използва.

3.1 Server

Ядрото на Strawberry prolog предоставя на COM клиентите един единствен обект – Server, който реализира един входящ интерфейс за управление - IStrawberrySvr и един изходящ интерфейс за събития – _IStrawberrySvrEvents. И двата интерфейса наследяват IDispatch, за да могат скриптовите езици (а и самия Strawberry prolog) да го ползват.

Самото приложение инициализира фабриката за класовете си, така че да позволява създаването на най-много един екземпляр на обект. При конструирането си обектът проверява дали приложението е стартирано директно (а не поради заявка за създаване на COM обект) и ако е така изкуствено създава обект, така си гарантираме, че това приложение няма да бъде използвано за COM. При последното освобождаване се прави отново проверка дали не е стартирано директно приложението и после изрично се затваря, като се изпраща съобщение за затваряне към главния му прозорец.

Самият обект може да се използва директно от Strawberry prolog по следния начин:

```
?-
    X is server("Strawberry.Server.1"),
    close(X).
```

3.2 IStrawberrySvr

Дефиницията на входящ интерфейс за управление е:

```
interface IStrawberrySvr : IDispatch
{
    [propget, id(1)]
    HRESULT program([out, retval] BSTR* pVal);
    [propput, id(1)]
    HRESULT program([in] BSTR newVal);

    [id(2)]
    HRESULT compile([out,retval] LONG* plResult);

    [propget, id(3)]
    HRESULT predicate([out, retval] BSTR* pVal);
    [propput, id(3)]
    HRESULT predicate([in] BSTR newVal);

    [propget, id(4)]
    HRESULT arguments([out, retval] BSTR* pVal);
    [propput, id(4)]
    HRESULT arguments([in] BSTR newVal);

    [id(5)]
    HRESULT call([out,retval] LONG* plResult);

    [propget, id(6)]
    HRESULT result([out, retval] VARIANT* pVal);
    [propput, id(6)]
    HRESULT result([in] VARIANT newVal);
}
```

Типичното му е използване е:

1. Създава се екземпляр на COM обекта **Server**.
2. Задава се пълното име на пролог програмата, която ще се изпълнява, попълвайки свойството му **program**.
3. Подготвя се за изпълнение посредством извикването на метода му **compile**.
4. Задават се цел и аргумента й посредством свойствата **predicate** и **arguments**.
5. Предава се управлението на пролог ядрото, извиквайки метода **call**.
6. Прочита се резултата (незадължителна стъпка) от свойството **result**, което има за стойност – стойността на глобалната променлива G_Return.
7. Освобождава се COM обекта.

Пример:

Нека имаме следната пролог програма с име „inc.pro“:

```
inc(X):-
    G_Return := X +1.
```

Програмата на пролог, която използва горната (и се намира в същата папка) е:

```
?- client_test.

client_test :-
    X is server("Strawberry.Server.1"),
    X.program:= get_current_directory()
        + "\\inc.pro",
    X.compile(),
    X.predicate:="inc",
    X.arguments:="3+2",
    X.call(),
    Result is X.result,
    write("result is "),
    write(Result),nl.
```

Като резултат от изпълнението й в изходния прозорец ще се появи:

```
result is 6
```

Разбира се, клиентската програма може и да не е на пролог, ето примерна JScript програма:

```
var X = WScript.CreateObject(
    "Strawberry.Server.1");
X.program = "_PUT_FOLDERHAME_HERE_"
    + "\\inc.pro";
X.compile();
X.predicate="inc";
X.arguments="3+2";
X.call();
var Result = X.result;
WScript.StdOut.WriteLine(
    "result is " + Result );
```

3.3 _IStrawberrySvrEvents:

Дефиницията на изходящ интерфейс за събития е:

```
dispinterface _IStrawberrySvrEvents
{
    HRESULT RaiseTheEvent(
        [in] VARIANT Val1,
        [in] VARIANT Val2,
        [in] VARIANT Val3,
        [out, retval] VARIANT* pvRes);
}
```

Strawberry prolog има вградена функция **raise_event**, която уведомява всички абонирани слушатели, че е настъпило събитие. Тя е с 3 аргумента, свободен формат и връща стойност – върнатата стойност от последния слушател, обработил събитието (това поведение може да се промени в бъдеще).

Пример:

Нека леко променим предишната „inc.pro“ пролог програма:

```
inc(X):-
    raise_event("inc", "begin", _),
    G_Return := X +1,
    raise_event("inc", "end", _).
```

Клиентската програма на пролог е:

```
?- client_test.

client_test :-
    X is server2("Strawberry.Server.1",on_event),
    X.program:= get_current_directory()
        + "\\inc.pro",
    X.compile(),
    X.predicate:="inc",
    X.arguments:="3+2",
    X.call(),
    Result is X."result",
    write("result is "),
    write(Result),nl.

on_event::M(|L):-
    write("on event "),
    write(L), nl.
```

Като резултат от изпълнението й в изходния прозорец ще се появи:

```
on event ["inc","begin",grayed]
on event ["inc","end", grayed]
result is 6
```

За да илюстрираме връщането на резултат ще променим двете програми, така че при командата за изчисление сървърът да пита клиента с колко да увеличи аргумента си.

Променяме „inc.pro“:

```
inc(X):-
    Y is raise_event("inc", X, "by"),
    G_Return := X + Y.
```

Съответната клиентска програма е:

```
?- client_test.

client_test :-
    X is server2("Strawberry.Server.1",on_event),
    X.program := get_current_directory()
        + "\\inc.pro",
    X.compile(),
    X.predicate:="inc",
    X.arguments:="3+2",
    X.call(),
    Result is X.result,
    write("result is "),
    write(Result),nl.


on_event::M(|L):-
    write("on event "),
    write(L), nl,
    G_Return:=3.
```

И резултатът е:

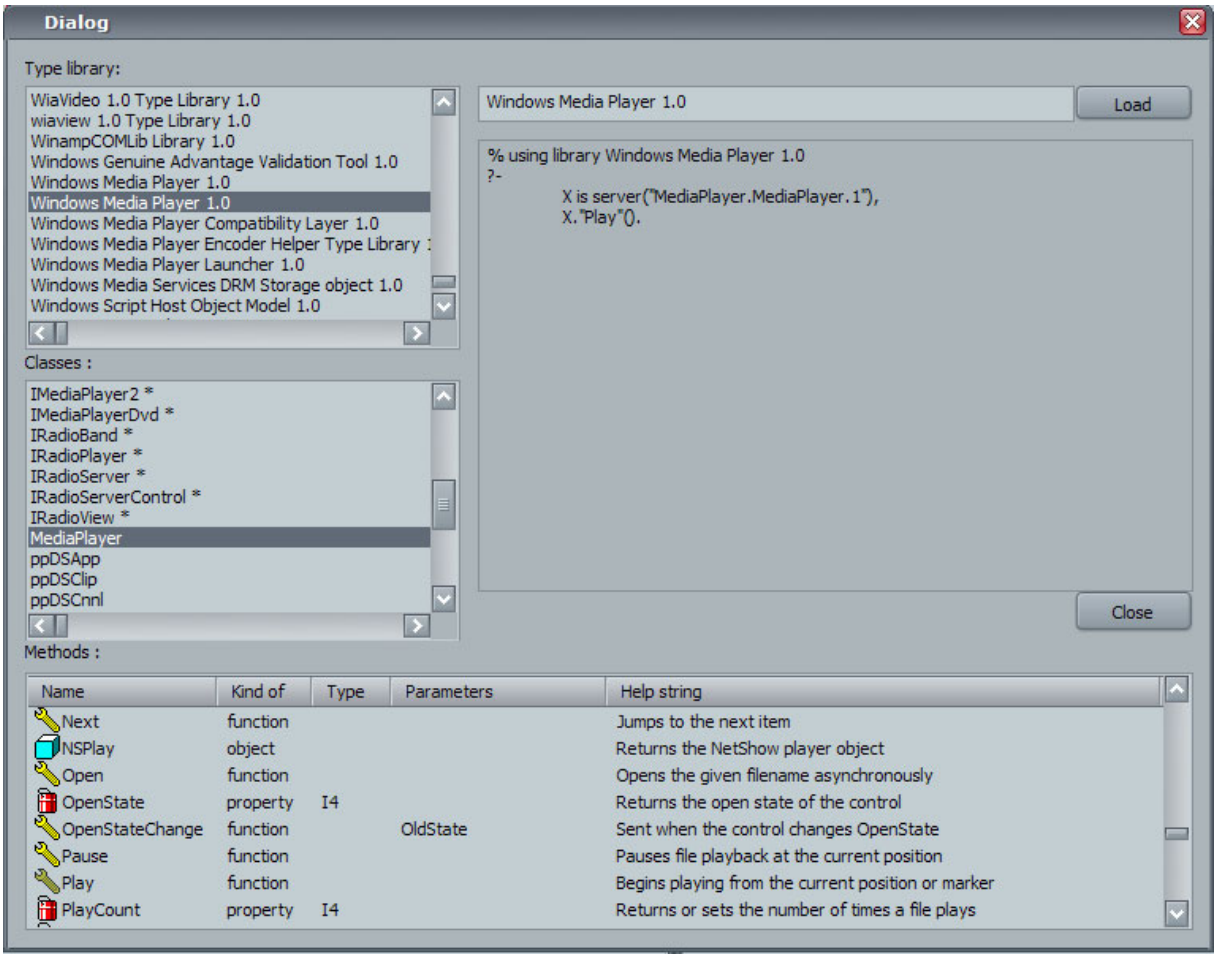
```
on event "event"(,"inc",5,"by")
result is 8
```

Отново не е задължително клиента да е на пролог – ето примерна програма на C# със същата функционалност (Strawberry е добавен като COM референция към C# проекта):

```
class Program
{
    static object onevent(
        object a, object b, object c)
    {
        System.Console.WriteLine(
            "on event "
            + a + ", " + b + ", " + c);
        return 3;
    }

    static void Main(string[] args)
    {
        Strawberry.StrawberrySvrClass X =
           new  Strawberry.StrawberrySvrClass();
        X.@event += new Strawberry.
_IStrawberrySvrEvents_RaiseTheEventEventHandler(
            onevent);
        X.program = "FOLDERHAME" +"\\inc_by.pro";
        X.compile();
        X.predicate = "inc";
        X.arguments = "3+2";
        X.call();
        object Result = X.result;
        System.Console.WriteLine(
            "result is " + Result);
    }
}
```

4 Съветникът Object Browser

Повечето среди за разработка имат съветници, подпомагащи разработчиците за рутинни задачи. Такъв е и **Object Browser**, който може да се активира от менюто Tools, ActiveX Browser. Във версиите до 2.5 неговата реализация беше: да се изброят всички регистрирани сървъри, да се създаде екземпляр на искания COM сървър и да се изброят методите и свойствата му (което както видяхме от примера с WORD не е безопасно). В новата версия съм пренаписал този съветник, така че той вече не работи със самите обекти, а с техните библиотеки на типовете (TLB – type library, което представлява компилиран IDL). Тези библиотеки са независима част от сървъра и биха могли и да липсват, но професионално направените сървъри винаги имат такава библиотека, която влиза в инсталацията им. Разработчикът може да разгледа регистрираните библиотеки или да зареди собствена. После да види обектите, които са описани в нея, с техните методи и свойства. И накрая да вземе примерен код, илюстриращ използването им.



В лявата колона в горното списъчно поле се показват всички регистрирани библиотеки (прочитат се от регистъра от HKCR/TypeLib) и се зарежда името, версията и някои други техни свойства.

При избор на библиотека, във второто списъчно поле от всички типове, дефинирани в нея, се показват само тези, които са обекти или управляващи интерфейси (типове от вид TKIND_COCLASS или TKIND_DISPATCH). Попълва се и полето за име на библиотеката.

Когато се избере типа в долния списъчен изглед се зареждат всички негови свойства и методи с техните аргументи. Показват се техните имена и описания. В дясното поле се генерира примерна програма, показваща как да се създаде избраният обект.

При избор на метод или свойство се допълва дясното поле с код как да се използва. На разработчика остава да копира кода и да го използва в собствената си програма.

Примерно, за да напишем програма на Strawberry prolog, която да прави резервно копие на кода си, ще стартираме съветника ActiveX Browser и последователно избираме:

1. Библиотека Windows Script Host Object Model 1.0.
2. Обект FileSystemObject.
3. Метод CopyFile.

Генерира се следния код:

```
% using library WScript Host Object Model 1.0
?-
    X is server("Scripting.FileSystemObject"),
    X.'CopyFile'(SOURCE,DESTINATION).
```

Копираме го, затваряме съветника и създаваме нова програма, в която го вмъкваме. Проверяваме, че програмата се компилира и добавяме „бизнес логиката“ на приложението:

```
% using library WScript Host Object Model 1.0
?-
    SOURCE :=
        this_program_path +
        "\\" + this_program_name,

    DESTINATION :=
        this_program_path +
        "\\" + this_program_name +
        "." + current_time(),

    X is server("Scripting.FileSystemObject"),
    X.'CopyFile'(SOURCE,DESTINATION).
```

Лекотата, с която създадохме горния пример е леко заблуждаваща – всъщност предварително знаехме кой обект ще използваме, каква функционалност има и от коя библиотека е, а съветникът просто свърши останалото. От друга страна е едно добро средство за запознаване с дадена библиотека, особено при липса на документация за нея. Употребата му драстично намалява вероятността от синтактични грешки – примерно при изписването на имената на методите.

5 Примери

Дотук бе показано:

1. Как да управляваме WORD (3.1.2).
2. Как програмата ни да бъде интерактивна с помощта на MS Agent (3.1.3).
3. Как можем да създадем браузър (3.1.4).
4. Как да се извикаме произволна пролог програма от JScrip и C# (4.X).

В тази глава ще дададем още примери за новите възможности, които се откриват пред пролог програмистите, ползващи COM технологията. Те са пренесени от мен от реални приложения, написани на други езици.

5.1 Работа с бази от данни

Използвайки **ADODB** имаме достъп до всички съвременни бази от данни.

Нека примерно имаме база на ACCESS, с таблица [country] с две колони: [country_code] – NUMBER и [country_name] – TEXT. Кодът, който прочита таблицата изглежда така:

```
?-    getCnnString(STR_Cnn),
      open_db(STR_Cnn,Obj_cnn),
      open_rs(Obj_cnn,Obj_rs),!,
      read_all(Obj_rs).


getCnnString(
"Provider=Microsoft.Jet.OLEDB.4.0;Data Source="
      + get_current_directory()
      + "\\db.mdb").



open_db(STR_Cnn,Obj_cnn) :-
      Obj_cnn is server("ADODB.Connection"),
      Obj_cnn.'ConnectionString' :=  STR_Cnn,
      Obj_cnn.'open'(),!.


open_rs(Obj_cnn,Obj_rs) :-
      Obj_rs is Obj_cnn.'execute'(
          "select * from country"),!.


read_all(Obj_rs) :-
      B_EOF is Obj_rs.'EOF',
      not(B_EOF),
      process_line(Obj_rs),
      Obj_rs.'MoveNext'(),
      read_all(Obj_rs),!.

read_all(_).
```

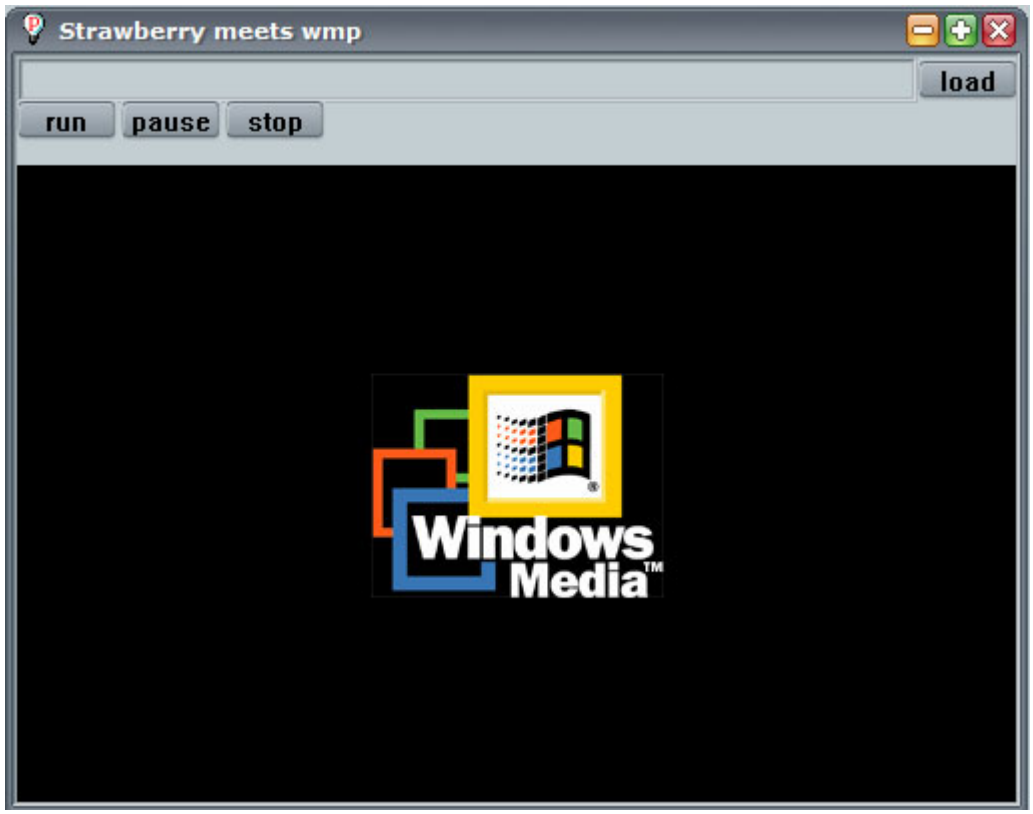
```
process_line(Obj_rs) :-
    Obj_Fields is Obj_rs.'Fields',

    Obj_Field_CountryCode is
        Obj_Fields.'Item'(
            "country_code"),
    N_CountryCode is
        Obj_Field_CountryCode.'value',

    Obj_Field_CountryName is
        Obj_Fields.'Item'(
            "country_name"),
    STR_CountryName is
        Obj_Field_CountryName.'value',

    write(
        N_CountryCode
        + " - "
        + STR_CountryName),nl.
```

Ако трябва да се прочетат същите данни от таблица на EXCEL трябва само да се смени връзката:

```
getCnnString(
"Provider=MSDASQL.1;Extended Properties=\"DBQ="
+get_current_directory()
+"\\country.xls;Driver={Microsoft Excel Driver
(*.xls)};\"").
```

5.2 Media Player

За да направим плейър ще използваме библиотеката **Windows Media Player**.




```
?- window( G_W, _, wnd_func(_),
     "Strawberry meets wmp", 10, 100, 511,406).
```

```
wnd_func(init) :-
  G_Obj is server("MediaPlayer.MediaPlayer"),
  G_Obj.'ShowControls' := 0,
  G_Obj.'AutoRewind' := 1,
  G_Obj.'AutoStart' := 0,
  G_Obj.'Mute' := 0,
  embed_server( G_WM, _, _,
    "WMP", 0, 54, 500,320, G_Obj),
  edit( G_EDIT_PATH, _, edit_func(_),
    "", 0, 0, 450, 23),
  button( G_BTN_Load, _, but_func_load(_),
    "load", 450, 2, 50, 20),
  button( G_BTN_Run, _, but_func_run(_),
    "run", 0, 22, 50, 20),
  button( G_BTN_Pause, _, but_func_pause(_),
    "pause", 52, 22, 50, 20),
  button( G_BTN_Stop, _, but_func_stop(_),
    "stop", 104, 22, 50, 20).

but_func_load(press) :-
    X is get_text(G_EDIT_PATH),
    G_Obj.'Open'(X),
    write("will play "),write(X),nl.

but_func_run(press) :-
    G_Obj.'Play'().

but_func_stop(press) :-
    G_Obj.'Stop'(),
    G_Obj.'CurrentPosition' := 0.

but_func_pause(press) :-
    G_Obj.'Pause'().
```

5.3 Работа с XML

С помощта на **MSXML** това е лесно, дори е възможно да консумираме web services. Ето примерно как да прочетем водещите новини на YAHOO:

```
    ?-
            SourceFile is
"http://rss.news.yahoo.com/rss/topstories",

            StyleFile is
                get_current_directory()
                + "\\rss.xsl",

            OutputFile is
                get_current_directory()
                + "\\rss.htm",


        % Load the XML
            Source is server("MSXML2.DOMDocument"),
            Source.'async' := 0,
            Res1 is Source.'load'( SourceFile ),
            Res1,

        % Load the XSLT
            Style is server("MSXML2.DOMDocument"),
            Style.'async' := 0,
            Res2 is Style.'load'( StyleFile ),
            Res2,

        % make transformation
            X is Source.'transformNode'(Style),

        % clean up XML/XSL
            close(Style),
            close(Source),
```

```
        % save result to disk
            FSO is server(
                "Scripting.FileSystemObject"),
            F is FSO.'OpenTextFile'(
                OutputFile,2,1,-2),
            F.'Write'(X ),
            F.'Close'(),

        % clean up FSO/F
            close(F),
            close(FSO),

        % open resulting html in default browser
        shell_execute(OutputFile),


        nl,write("END"),nl.
```

Където „rss.xsl“ е примерно:

```
    <?xml version="1.0"?>
    <xsl:stylesheet
        xmlns:xsl=
         "http://www.w3.org/1999/XSL/Transform"
        version="1.0">
        <xsl:output method="html"/>
        <xsl:template match="/">
            <ul>
            <xsl:for-each select="//item">
<li>
    <xsl:value-of select="pubDate"/>
    <a>
        <xsl:attribute name="href">
            <xsl:value-of select="link"/>
        </xsl:attribute>
        <xsl:value-of select="title"/>
    </a>
</li>
            </xsl:for-each>
            </ul>
        </xsl:template>
    </xsl:stylesheet>
```

6 Какво може да бъде добавено

В следващите реализации към ядрото може да бъде добавена поддръжка на асинхронни извиквания от COM клиентите – примерно клиента извиква метода call и вместо да бъде блокиран изчаквайки резултат от изпълнението му, да бъде уведомен със събитие on_call_end.

При необходимост ядрото и графичният модул на Strawberry prolog могат да бъде разширени, така че да се реализира функционалността на OLE сървър, позволяваща на самия пролог или на негови прозорци да бъдат вмъквани като визуална компонента в други приложения.

Също така може да се наложи работа с COM обекти, които не предоставят директно IDispatch, но от техния IUnknown може да се вземе друг интерфейс, който вече е наследник на IDispatch. Конкретно такъв проблем има, ако искаме да има поддръжка на DirectX (и по специално на DirectMedia).

7 Библиография

[1] Inside Visual C++ 5.0

David J,Kruglinski

© 1998 Microsoft Corporation

[2] Active Xpert

Tom Armstrong

with Jim Crespino and Rob Alumbaugh

© McGraw-Hill Companies,Inc.1999

[3] October 2005 Release of the MSDN Library

online : http://msdn.microft.com/library

[4] COM: Handle Late-bound Events within Visual Basic Using an ATL Bridge

Carlo Randone

March 2001 issue of MSDN Magazine

[5] Control Containment in ATL

By Fritz Onion

Published in C++ Report, July 1999 issue.

http://www.pluralsight.com/articlecontent/cpprep0799.htm

[6] Strawberry Prolog Help

версия 2.5

http://dobrev.com/

[7] Кратък английско-български терминологичен речник

Михаил Балабанов

http://m-balabanov.hit.bg/