1998 International Computer Symposium
Workshop on Artificial Intelligence
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Bring Intelligence to Object-Embedded Documents Using PrologScript Language

Tai-Wen Yue and Su-Chen Chiang

Department of Computer Science & Engineering
Tatung Institute of Technology

Abstract

Nowadays, it is not uncommon that a web page includes many ActiveX controls and/or applets, which are controlled via various scripts coming with the page. Besides reducing Internet traffic, this also makes the page more active and enhances software reusability. Two leading scripting languages are VBScript, which was provided by Microsoft, and JavaScript, which was provided by Netscape. Such procedural languages, however, are inappropriate in AI (artificial intelligence) programming. To apply AI technology into Web, we develop the so-called PrologScript language. This allows Web page designers to embed rules into documents and, as a result, make controls in documents, seemingly, intelligent.

This research is based on COM (Component Object Model) model and ActiveX Scripting technology. In the paper, we'll describe the architecture of a scripting engine and its counterpart, say, scripting host. In addition, we highlight the strategy of building a scripting engine using available programming environment in a local system. Two concrete examples are given to manifest the feasibility of using PrologScript to bring intelligence into object-embedded documents.

1. Introduction

At the beginning stage of the World Wide Web (WWW), an HTML document is simply comprised from a set of hypertext information [2]. Today, due to the rapid evolution of object technology, most web browsers can also render objects or controls, which carry execution code to the client site to perform certain tasks while a HTML document is downloaded. The popular controls currently used in WWW are Java applets and ActiveX controls.

To achieve reusability and flexibility, controls are generally designed programmable. Specifically, a control includes a set of properties to configure its profile, a set of methods to control its behavior, and itself can generate events to feedback its state to listeners. Henceforth, a scripting mechanism is required to coordinate the behavior of such embedded controls. Today, the two leading scripting languages for writing scripts are JavaScript [10] and VBScript [8]. These two scripting languages are syntactically very similar to one another, and are Algol-type procedural languages [1]. To enrich the scripting capability, it's conceivable that other types of language might be developed so as to achieve specific goals easier.

Although JavaScript and VBScript are sufficiently powerful in many situations, such procedural languages, however, are hardly to let us bind intelligence into a document. In the paper, we discuss the newly developed scripting language called PrologScript. PrologScript is made up of a subset of Prolog language for scripting. The corresponding scripting engine is developed using *ActiveX Scripting* technology [7], which is defined based on COM [9, 12] object model.

Since the WWW comes to our life, office automation heavily depends on using Internet and/or Intranet applications. Many such applications in WWW may also be required to intelligently interact with users, such as stock analysis, decision making. Prolog is a rule-based language. It is particular suitable for AI programming, e.g., building an expert system [4]. To build an AI application, the programmer simply describes the logic among all of its terms, including atoms, variables, lists, ... etc., using clauses (rules). This, hence, will ease and accelerate the design process for AI applications (when compared with using traditional Algol-type languages.)

This paper is organized as follow: Section 2 gives a brief review of the COM object model and describes the architecture of a scripting engine. Section 3, describes the syntax and semantics in writing a PrologScript. Section 4 discusses the main components for the scripting host and scripting engine of PrologScript. In Section 5, two games, called slide-block-puzzle and 3-pile-nim, are demonstrated to manifest the applying of PrologScript to AI applications. Finally, we draw a conclusion in Section 6.

2. COM And ActiveX Scripting

2.1 COM

The core of the Component Object Model is a specification for how components and their clients interact. For any platform, COM defines binary standard for interoperability of software components. An object creates a *vtable*, see Figure 1, that contains pointers to the implementations of the interface member functions. The client's pointer to an interface is, in fact, a pointer to the pointer of the vtable. Thus, components can be implemented in any programming languages and used by clients that are written using completely different programming languages.

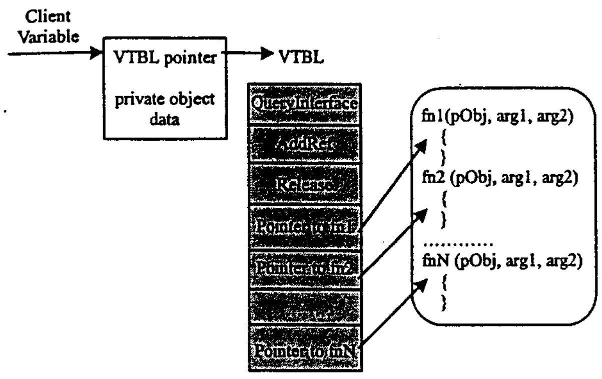

Figure 1. Virtual function tables (VTBL)

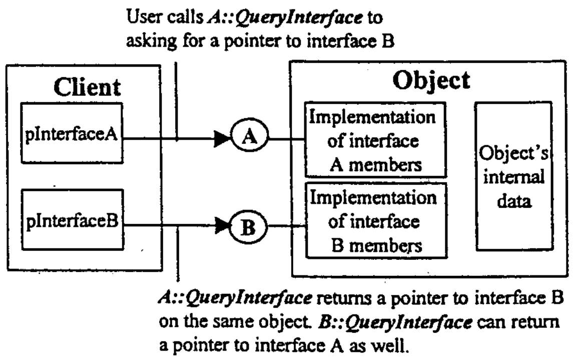
User calls A::QueryInterface to asking for a pointer to interface B
Client
pInterfaceA
pInterfaceB
A
B
Object
Implementation of interface A members
Implementation of interface B members
Object's internal data
A::QueryInterface returns a pointer to interface B on the same object. B::QueryInterface can return a pointer to interface A as well.

Figure 2. Using *IUnknown::QueryInterface*

Code 1. The members of *IUnknown* interface.

```
interface IUnknown
{
  HRESULT QueryInterface(REFIID riid,void** ppvObject);
  ULONG AddRef();
  ULONG Release();
}
```

To become a COM object, the object at least must implement the *IUnknown* interface. It has three member functions as shown in Code 1. Furthermore, COM allows an object to implement multiple interfaces, all of them have to inherit the *IUnknown* interface. The *IUnknown::QueryInterface()* member function allows the client to perform interface navigation, i.e., the client can navigate to any interface from anywhere by issuing *QueryInterface()* request on the underlying interface given the goal interface ID as an input parameter, as shown in Figure 2. In addition, the object must maintain a reference counter to keep track its interface usage. Specifically, if the object reports an interface to the client (i.e., the object copies the pointer of a vtable to a client's location), it must call *AddRef()* once to increase the reference count. As for the client, if an interface is no more used, the client must call the *Release()* member function once on that interface. Upon receiving the release call, the object will decrease the reference count. If the counter reaches zero, the object will destroy itself. This means that the object will automatically return its resource to the system if it is not used by any client.

In summary, with COM, an object is a piece of compiled code that provides some service to the rest of the system. It is a binary standard of how you talk to an object, how that object handles its own lifetime, and how it tells the world that it can do.

2.2 ActiveX Scripting

Programs that drive other applications are sometimes called scripts. In other words, script is the data that make up the program that the scripting engine runs. You could place an ActiveX control on a page, but it may work improperly without script. ActiveX controls are actually more reliant on scripting than other kinds of Web interface elements. We will take a look at ActiveX Scripting, which is an OLE [3] communication technology rather than a scripting language.

With ActiveX Scripting, hosts can call upon disparate scripting engine from multiple sources and vendors to perform scripting between components. The implementation of the script itself—language, syntax, persistent format, execution model, and so on—is left to the script vendor. The purpose of this standard is to define a method for a scripting host to call on various scripting engines and allow communication between objects within an OLE container. Three different elements, see Figure 3, are involved in an ActiveX Scripting session: the ActiveX Scripting host, an ActiveX Scripting engine, and the window, called container, containing the code and controls.

An ActiveX Scripting host is a piece of code that communicates with scripting engine(s). When you define a script, it's the host that accepts it and then sends the commands to the engine. A scripting host typically hosts objects, placed in a container, with their methods, properties, and events can be invoked, accessed, and received by an executing script. The most common example of an ActiveX scripting host right now is Internet Explorer 4.x. A script is executed by a scripting engine under the control of a host. An ActiveX scripting engine is the object that actually interprets the script. There are no limitations on the precise language syntax or even the form of the script. ActiveX Scripting engines can be developed for any language or run-time environment, including Microsoft Visual Basic for Application (VBA), Microsoft Visual Basic Scripting Edition (VBScript), Perl, and Lisp. In the next section, we'll describe how to apply the architecture to develop PrologScript engine.

2.3 Interfaces and Executing Scenario

The main interfaces defined for ActiveX Scripting are summarized as follows:

Scripting Host:

- *IActiveScriptSite*: The host must create a site for the engine's communication by implementing this interface. It monitors events such as the starting and stopping of scripts and when a script error occurs.

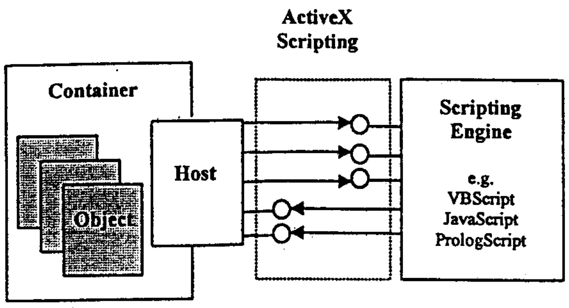

Figure 3. Three different elements that are involved in an ActiveX Scripting session

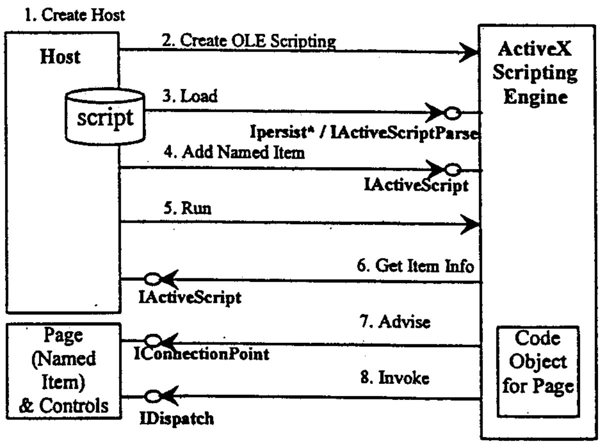

Figure 4. ActiveX Scripting Basic Architecture

Code 2(a): The script of Polygon control with VBScript

```
<HTML>
<HEAD>
<TITLE> PolyCtl with VBScript</TITLE>
</HEAD>
<BODY>
<OBJECT ID="polygon"
  < CLASSID=
    "CLSID:4CBBC676-507F-11D0-B98B-000000000000"
  >
>
</OBJECT>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub polygon_ClickIn(x, y)
    polygon.Sides = polygon.Sides + 1
End Sub
Sub polygon_ClickOut(x, y)
    polygon.Sides = polygon.Sides - 1
End Sub
-->
</SCRIPT>
</BODY>
</HTML>
```

Code 2(b): The script of Polygon control with PrologSript

```
<HTML>
<HEAD>
<TITLE> PolyCtl with PrologScript</TITLE>
</HEAD>
<BODY>
<OBJECT ID="polygon"
  < CLASSID=
    "CLSID:4CBBC676-507F-11D0-B98B-000000000000"
  >
>
</OBJECT>
<SCRIPT LANGUAGE="PrologScript">
<!--
polygon(onClickIn, X, Y):-
    polygon(getSides, P1),
    P2 is P1+1,
    polygon(setSides, P2).

polygon(onClickOut, X, Y):-
    polygon(getSides, P1),
    P2 is P1-1,
    polygon(setSides, P2).
-->
</SCRIPT>
</BODY>
</HTML>
```

- *IActiveScriptSiteWindow*: The host can support this interface to allow an engine access to that object's window, e.g., displaying a message window.
- *IProvidMultipleClassInfo*: This interface provides access to the type information of the default interfaces that describe an extended object. A container typically implements an extender object to add properties, methods, and events to an existing object, extendee.

Scripting Engine:

- *IActiveScript*: Every scripting engine must support this interface. A user uses the method in *IActiveScript* to pass the engine a pointer to the host's *IActiveScriptSite* interface, to tell the script begin executing and performing other tasks.
- *IActiveScriptParse*: Scripting engines that allow script text to be added dynamically can support this interface.

The steps involved in the interaction between the host and engine [8], as shown in Figure 4, are as follows:

1. The host loads a document, which may designate the script language to be used for the associated script, for example, in the LANGUAGE attribute of <SCRIPT> tag, see Code 2.
2. The host creates a scripting engine by treating the value of that attribute as the engine's *ProgID*. A call to *CoCreateInstance()* completes the task.
3. The host calls engine's *IActiveScriptParse::InitNew()* to create an empty script and, then, calls *IActiveScriptParse::ParseScriptText()* to set the script context into the engine.
4. The host calls the *IActiveScript::AddNamedItem* method to add a top-left named item into the engine's name space.
5. Now, ActiveX Scripting engine has everything it needs to run the script; the host issues an

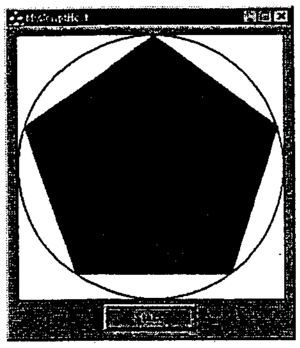

Figure 5. The Polygon Control

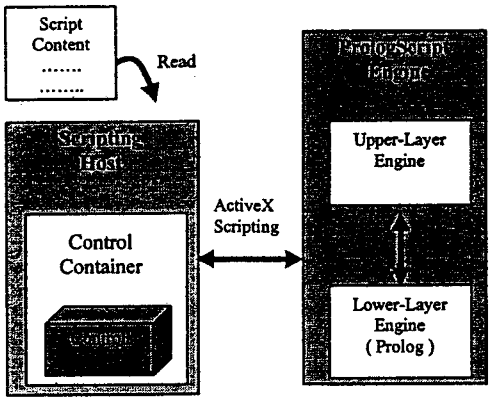

Figure 6. PrologScript engine Architecture

IActiveScript::SetScriptState(SCRIPTSTATE_CONNECTED) call to start the script.

6. Each time the script engine need to associate a symbol with a top-level item, it calls the *IActiveScriptSite::GetItemInfo* method, which returns information about the given item.
7. Before starting the actual script, the scripting engine connects to the events of all the relevant objects through the *IConnectionPoint* interface.
8. As script runs, the scripting engine realizes references to methods and properties on named objets through *IDispatch::Invoke* or other standard OLE binding methods.

3. The PrologScript

PrologScript is a scripting language formed from a subset of Prolog language. In essential, the syntax of PrologScript is similar to that of Prolog. In the following, we informally describe the syntax regarding to control's operations. We assume the object being named `theObject`.

3.1 get/set Properties

To get/set a property, say, `Property` of `theObject`, use the following syntax:

Code 3. The Type Information of a Polygon object

```
import "oaidl.idl";
import "ocidl.idl";
[
 object,
 uuid(4CBBC675-507F-11D0-B98B-000000000000),
 dual
]
interface IPolyCtl : IDispatch
{
  [propget, id(1)] HRESULT Sides([out, retval] short
                                              *newVal);
  [propput, id(1)] HRESULT Sides([in] short newVal);
};
[ uuid(4CBBC673-507F-11D0-B98B-000000000000) ]
library POLYGONLib
{
  importlib("stdole32.tlb");
  [ uuid(4CBBC677-507F-11D0-B98B-000000000000) ]
  dispinterface _PolyEvents
  {
    properties:
    methods:
    [id(1)] void ClickIn([in]long x, [in] long y);
    [id(2)] void ClickOut([in]long x, [in] long y);
  };
  [ uuid(4CBBC676-507F-11D0-B98B-000000000000) ]
  coclass PolyCtl
  {
    [default] interface IPolyCtl;
    [default, source] dispinterface _PolyEvents;
  };
};
```

```
Get Property : theObject(getProperty, X)
Set Property : theObject(setProperty, Y)
```

That is, to get/set a property, one simply appends the property name defined in IDL (interface description language) of the object after "get"/"set". To get a property, `X` must be an unbounded variable (in Prolog, an unbounded variable begins with a capital character). To set a property, `Y` must be a bounded variable or a literal (such as a digital value, or a string.)

3.2 Method Invocation

To invoke a control's method, say, `Method` with parameters, say, `(<parml>, <parm2>, ...)`, use the following syntax:

```
Method Invocation : theObject(callMethod,
<parml>, <parm2>, ...)
```

That is, to invoke a method, one simply appends the method name defined in IDL after "call". For each `[in]` parameter of a method, it must be passed as a bounded variable or a literal. Conversely, for each `[out]` parameter, it must be passed as an unbounded variable. For each `[inout]` parameters, for example, `X`, it must be passed as an order pair `[Xin, Xout]`, i.e., a two-element list, where `Xin` must be a bounded variable or a literal while `Xout` an unbounded one.

3.3. Event Handlers

To define a event handler for an event, say, `Event` with parameters, say, `(<parml>, <parm2>, ...)`, use the following syntax:

```
Event Handler : theObject(onEvent, <parml>,
<parm2>, ...)
```

That is, to define an event handler, one simply appends the

Code 4. The code of *invoke* rule

```
invoke(Obj, Name, Param):-
  atom_string(Obj, SObj), string_chars(SObj, LObj),
  append(LObj, [44], L1),
  atom_string(Name, SName), string_chars(SName, LName),
  append(L1, LName, L2),
  (
    (
      ( ( number(Param), number_string(Param, SParam) );
        ( atom(Param), atom_string(Param, SParam) );
        ( string(Param), SParam=Param)
      ),
      append(L2, [44], L3),
      string_chars(SParam, LParam),
      append(L3, LParam, Last),
      string_chars(S, Last)
    );
    ( var(Param), string_chars(S, L2))
  ),
  window(Handle),
  sendMsg(Handle, 1025, 0, 0, S),
  (
    ( var(Param), sendMsg(Handle, 1026, 0, 0, Param), !);
     sendMsg(Handle, 1026, 0, 0, P), !
  ), !.
```

event name defined in IDL after "on". The parameter passing convention is the same as that defined for method invocation.

3.4. The Comparison between PrologScript and VBScript

For comparison reason, we use a simple ActiveX Control called *Polygon*, see Figure 5, to highlight the differences between PrologScript and VBScript. The IDL of Polygon control is listed in Code 3. The control has a property named `Sides`. By varying its value, the control will redraw the polygon with the specific number of sides. Figure 5 shows the polygon with `Sides` being set to 5. The control also generates two events `ClickIn` and `ClickOut`, which are fired while mouse left button is clicked inside and outside the polygon respectively. The event handlers written using PrologScript and VBScript are listed in Code 2.

4.The Architecture

Figure 6 shows the architecture of scripting host and engine for PrologScript. The interfaces between the host and engine follows the ActiveX Scripting described in Section 2.

4.1 The Scripting Host

Any browsers that follow the specification of the ActiveX Scripting, for example, IE 4.0, can serve as a scripting host of the proposed PrologScript engine. For experiment purpose, we have built a simplified host using Visual C++. Our host only eats contents located inside <OBJECT> and <SCRIPT> tags in HTML documents. The host also serves as a control container.

Upon receiving a document, e.g., see Code 2. The host performs the following parsing actions:

1. When encounters <SCRIPT> tag, the host extracts the engine's ProgID described in LANGUAGE attribute, e.g., "PrologScript", "VBScript" or "JavaScript". Accordingly, the host retrieves the engine's CLSID by calling *CLSIDFromProgID()*. Then, the host calls *CoCreateInstance()* to launch the engine and, through the invocation of *IActiveScript::SetScriptSite()*, passes the *IActiveScriptSite* interface to the engine to make that as an connection channel from engine to host. Furthermore, the host calls *IActiveScriptParse::InitNew()* to initialized an empty script context. The rest of content in this tag, i.e., script context, is then sent to the engine by issuing *IActiveScriptParse::SetScriptText()*.
2. When encounters <OBJECT> tag, the host extracts its ID and CLSID attributes to serve as the object name and to create the object, respectively. The object name (becomes a property) and its associated *IDispatch* is then put into a type library, which is created for extending objects.
3. When reaches the end of the document, the host calls the *IActiveScript::AddNamedItem()* to add its top-level named item to engine's name space. This enables the engine to dynamically retrieve the *IDispatch* interfaces of extending objects.
4. The host now is ready. It then starts the engine by setting the engine's state to SCRIPTSTATE_CONNECTED by using *IActiveScript::SetScriptState()* call.
5. As an ending, the host closes the engine by calling *IActiveScript::Close()*, and releases all the interfaces it holds.

If the steps from 1 to 5 are successfully done, the host and the engine both are ready to interact one another.

4.2 The Scripting Engine for PrologScript

This prototype of scripting engine is built on the top of an existing Prolog product WIN-PROLOG [5], developed by Logic Programming Associates LTD. For convenience, we call the *upper-layer* engine shown in Figure 6 as engine, and call the *lower-layer* engine as WIN-PROLOG unless otherwise specified. Because WIN-PROLOG supports DDE (Dynamic Data Exchange protocol) [11], the engine and WIN-PROLOG communicates using the established DDE channel. Passing WM_DDE_POKE or WM_DDE_EXECUTE messages between these two layers, the engine can set data to WIN-PROLOG or request it to handle events, and vice versa.

The interfaces that we implement are consistent with those described in Section 2. The actions that the engine performs are described as follows:

1. As the instance of the engine is created, it seeks to find the WIN-PROLOG server using DDE protocol, i.e., broadcasts WM_DDE_INITIATE message for finding application named "Prolog Server", and with topic name being "Engine". If failed, we will notify the user to start up the server manually and to try it again. After connection established, the engine then sets itself to "Uninitialized" state and waits host's invocation.
2. Upon receiving *IActiveScript::SetScriptSite* and

IActiveScriptParse::InitNew invocations from the host (in "Uninitialized" state), the engine steps into "Initialized" state. In the scope of the invocations, the engine saves the host's *IActiveScriptSite* interface pointer and reports its state to the host by the invocation of *IActiveScriptSite::OnStateChange()*.

3. Upon receiving *IActiveScript::AddNamedItem* (in "Initialized" state), the engine adds this top-level item name into engine's name space.
4. Upon receiving *IActiveScriptParse::SetScriptText* (in "Initialized" state), the engine asks WIN-PROLOG to clear any script that it has ever received (via WM_DDE_EXECUTE message). Then, it hands this newly received script to WIN-PROLOG (via WM_DDE_POKE message.)
5. Upon receiving *IActiveScript::SetScriptState (SCRIPTSTATE_CONNECTED)* (in "Initialized" state), the engine, incorporating with the host, performs a sequence of preparation tasks (to be detailed shortly). After the preparation has suitably been done, the engine steps to "Connected" state and notifies the host this. Now, both the host and the engine are ready to talk to one another.
6. When the host wants to terminate the engine, it will notify the engine by the invocation of *IActiveScript::Close()*. Upon this, the engine sends WM_DDE_TERMINATE message to WIN-PROLOG to terminate the associated DDE channel and, then, releases any host-site interfaces it holds.

The most complicate part in writing engine's code is that to respond the *IActiveScript::SetScriptState (SCRIPTSTATE_CONNECTED)* invocation from the host. In the following, we summarize the tasks that should be done by the PrologScript engine to respond this call:

1. **Getting dispinterfaces of objects:** In order that the engine is able to get/set the properties, and to invoke methods of all objects hosted by the container (including the container itself), the engine has to acquire their dispinterfaces (a shorthand for *IDispatch* interface). To this, the engine, through the *IProvideMultipleClassInfo* interface provided by the host, queries the type library information that the host dynamically builds during parsing phase (refers to Step 2 that the host performs.) From the type information, a set of object names is obtained. By that, the engine can get the all related *IDispatch* interfaces of object instances hosted by the container.
2. **Building event sinks for objects:** Through the *IProvideClassInfo* interface of each object, the engine can get its type information. If the object can source events, it should include an entry of dispinterface with `[source]` attribute in the coclass module (see Code 3 for an example). If such an entry is available, the engine generates an event-sink object and makes a connection to the event source. The event-sink object has a dispinterface, which is dynamically built by consulting to the type information that we obtained earlier. The source-sink connection is established by negotiating the *IConnectionPointContainer* using the IID, which is prescribed in the type information, of the source dispinterface. When the source makes an invocation, the request will be delegated to WIN-PROLOG for service.
3. **Rules generation:** In a Prolog system, everything is driven by rule. Consider the event handler *polygon(onClickIn, X, Y)* shown in Code 2.

```
polygon(onClickIn, X, Y):-
  polygon(getSides, P1),
  P2 is P1+1,
  polygon(setSides, P2).
```

It is clear that there are three sub-goals in the above rules. If without corresponding rules that serve to achieve those sub-goals, this rule will be failed. This reveals that, although the scriptwriter does not write rules to perform property get/set or functional invocation, the engine automatically generates them for him/her. As a demonstration, given the type information of `Polygon` in Code 3, the engine generates the following rules:

```
polygon(getSides, NewVal):-
  invoke(polygon, getSides, Return),
  number_string( NewVal, Return).
polygon(setSides, NewVal):-
  invoke(polygon, setSides, NewVal).
```

It is not hard to see that these rules further have sub-goals. Sub-goal `number_string` is a WIN-PROLOG function, which performs conversion between a string and a numeric value. Sub-goal `invoke` is the one that we write to establish the connection from lower-layer engine (the WIN-PROLOG) to upper-layer engine (or simply as engine). The rule `invoke` is, somehow, cumbersome because low-level function call is involved. Basically, in the rule, we apply conversion functions (rules in WIN-PROLOG) to convert parameters and the return result(s), and use WIN32 low-level function call interface (support by WIN-PROLOG) to communicate with the engine through the established DDE channel. For clarity, we list the rule of `invoke` in Code 4.

5. Example AI Applications

In many problems, state space growth is combinatorially explosive, with the number of possible states increasing exponentially with the depth of the search. In AI, the well-known A* algorithm [6] is very efficient in defeating this combinatorial explosion and finding an acceptable solution. A* algorithm is a heuristic algorithm along with an evaluation function. Using Prolog to implement A*, its root pattern (rules), in general, is in the form of:

```
1. goal(Start, End) :-
   initialize(Start),   moveToGoal(Start,
    End).
2. moveToGoal(X, X):-!.
3. moveToGoal(From, To) :-
    chooseNextStep(NextStep),
    gotoNextStep(NextStep),
    moveToGoal(NextStep, To).
```

The user demands Prolog to find out a way to reach a goal by specifying the *Start* and *End* configurations corresponding to that goal, as shown in the first rule depicted above. This rule initializes an *Open* list and a *Close* list so as to keep track the paths that haven't and have been tried. It, then, starts its main task by invoking another sub-goal *moveToGoal(From, To)*. The rules to achieve this sub-goal are described in the 2nd and 3rd rules. It can be easily seen that the 2nd rule simply terminates the inference process of Prolog. The 3rd rule is the one that is most critically to effect the efficiency to solve a problem. Human intelligence, namely heuristic, can be appropriately involved to improve solution quality and performance. In particular, the rule for sub-goal *chooseNextStep(NextStep)* should be designed based on the so-called evaluation function. In terms of AI, the evaluation function corresponds to solution cost, may be a predicted one, to a problem. Therefore, the rules to reach this sub-goal are problem dependent. In the following, we'll apply different such rules to solve the corresponding problems. The next sub-goal, say, *gotoNextStep(NextStep)* is simply used to report the node being reached. And, the last sub-goal, say, *moveToGoal(From, To)* simply repeats the aforementioned scenario by recursively invoking the goal of itself.

5.1 The Game of Sliding-Block-Puzzle

The game is defined as follows: Consider a sliding block puzzle with the following initial configuration as shown in following:

0	1	2	3	4	5	6
W	W	W	E	B	B	B

There are three black tiles (B), three white tiles (W), and an empty cell (E). The puzzle has the following legal moves:

(a) A tile may move into an adjacent empty location.

(b) A tile can hop over one other tile into the empty cell.

The goal of this game is to make as few moves as possible to bring all of the black tiles to the left of all of the white tiles, and to make the empty cell divides two kinds of color tiles.

We've designed an ActiveX control, see Figure 7, to represent such a puzzle. The control has only one property named *Config* (the puzzle configuration) and without any method. While the automation controller changes its value, the control will redraw itself (in animation) to reflect this change.

With a little investigation, one can see that there are infinitely many paths that can lead puzzle into a goal state if solution cost is not considered. In many applications, the perfect knowledge to solve a problem is hardly available. In such a case, one may resort to use A* approach by specifying a reasonable evaluation function. To demonstrate using PrologScript to solve the problem, the evaluation function that we define is very simple. Given a configuration of the puzzle, by comparing it with the goal, the solution cost is defined by the number of mismatches between them. For example, the goal state of this puzzle is BBBEWWW, and the value of evaluation function of configuration BBEBWWW is 2 because the tiles at locations 2 and 3 are mismatch. Through this evaluation function you can decide which node, seemingly, will be the best for the next move. For comparison, an exhaustive approach is also applied to this game. The total steps resulted by using exhaustive search and A* are 151 and 22, respectively.

5.2 The 3-Pile-Nim

The sliding-block-puzzle is independent on user's interaction. Now, we demonstrate another game with user interaction being involved. The game, called 3-pile-nim, is defined as follows: There are three piles initially with different number of balls. Two players are required to play the game. They withdraw balls from those piles alternately. The restriction is as follows: For each move, one can withdraw any number of balls only from one of those piles. The one that makes all piles empty will lose the game.

The ActiveX control used to demonstrate this game is shown in Figure 8. The control has following properties: `Count1`, `Count2` and `Count3` represent the numbers of balls in the 1st, 2nd and 3rd piles, respectively, and `DownCount1`, `DownCount2`, and `DownCount3` represent the numbers of balls to be withdraw from the 1st, 2nd and 3rd piles, respectively. A user can set the values of those downcounters using the spin controls. Only one of the values can be nonzero, however.

In the game, one player is computer and another is a user. In order to win the game, the decision to be made should depend on the opponent's last move. With "Prolog" as a player, heuristics has to be embedded in rules to win the game. Because two players are involved in this game and the goal is clear, the 3rd rule described above is modified as follows:

```
3. moveToGoal(From) :-
    chooseNextStep(From, NextStep),
    gotoNextStep(NextStep).
```

That is, when turns to Prolog to make a new move, one simply invokes goal `moveToGoal` by specifying the current configuration of the piles. Therefore, we must design an evaluation function to make Prolog smart enough. Shamelessly to say, with the following heuristics, the polygon can beat any of my colleagues in the game. The rule is described as follows:

1. The one who makes two piles with equal number of balls and another pile empty will lose the game.

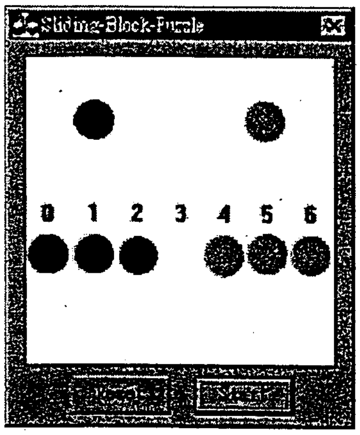

Figure 7. The game of sliding-block-puzzle

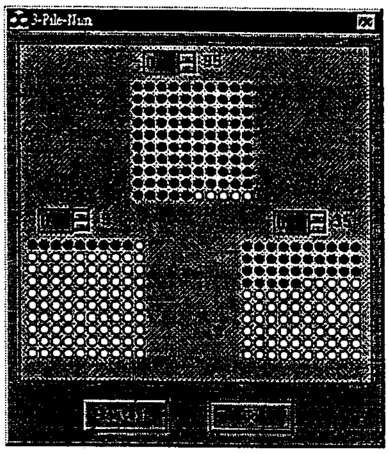

Figure 8. The game of 3-pile-nim.

2. The one who makes the three piles to become (1, 1, 1) will win the game.
3. The one who makes the three piles to become (1, 2, 3), (1, 4, 5), (1, 6, 7),..., and so on, will win the game.
4. The one who makes the three piles to become (2, 4, 6), (2, 5, 7), (2, 8, 10), (2, 9, 11),..., and so on, will win the game.

These rules can be easily coded using PrologSript.

6. Conclusion

ActiveX controls are ready-to-use elements. Therefore, recoding and recompilation are unnecessary to embed such components into documents. This reveals that useful controls are generally designed to be programmable and controllable. To function a set of controls to achieve a particular goal, a document designer scripts them using high-level scripting languages. Two well-known scripting languages are VBScript, and JavaScript. These languages are sufficiently powerful and found many uses, particularly, in Web-page design. In the paper, we propose the so-called PrologScript. The goal of the research is to provide control users another tool for programming controls using AI technology. With PrologScript, one can easily formulate the problem-solving heuristics into rules. Upon receiving user requests, Prolog inference engine then performs actions to carry out its corresponding tasks autonomously by firing the predefined rules.

Reference

[1] D. Appleby and J. J. Vandekopple, *Programming Languages: Paradigm and Practice* 2nd ed., McGRAW-HILL, pp. 100-113, 1997.

[2] T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret, "The World-Wide Web, " Comm. of the ACM, Vol. 37, No. 8, pp. 76-82, Aug. 1994.

[3] K. Brockschmidt, *Inside OLE* 2nd ed., Microsoft Press, 1995.

[4] J. Giarratano and G. Riley, *Expert Systems: Principle and Programming* 2nd ed., PWS, pp. 1-54, Aug. 1994.

[5] Logic Programming Associates LTD, *WIN_PROLOG - Personal Edition*, http://www.lpa.co.uk/per.html.

[6] G. F. Luger and W. A. Stubblefield, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving* 3rd ed., ADDISON-WESLEY, pp.140-144, Aug. 1994.

[7] Microsoft Corporation, *ActiveX Scripting*, Microsoft Developer Network (MSDN) Library, SDK Documentation, Platform SDK, ActiveX SDK, ActiveX Controls, April 1998.
http://www.microsoft.com/msdn/sdk/inetsdk/help/compdev/scripting/scripting.htm

[8] Microsoft Corporation, *VBScript*, Nov. 1998,
http://www.microsoft.com/scripting/vbscript/default.htm.

[9] Microsoft Corporation and Digital Equipment Coperation, *The Component Object Model Specification*, Draft Version 0.9, Oct. 24, 1995.
http://www.microsoft.com/oledev/olecom/title.htm.

[10] NetScape, *JavaScript*.
http://home.netscape.com/comprod/products/navigator/version_2.0/script/index.html.

[11] H. Rodent, "Supporting the Clipboard, DDE, and OLE in Applications," MSDN Tech. Group.
http://premium.microsoft.com/msdn/library/techart/msdn_ddeole.htm.

[12] *The Component Object Model: Technical Overview*, Dr. Dobbs Journal, Dec. 1994.
http://www.microsoft.com/com/wpaper/Com_modl.htm.