ISA Dialog Manager

OLE Interface

A.06.03.b

This manual describes the OLE interface of the ISA Dialog Manager which is available as an option of the IDM for Microsoft Windows. OLE (Object Linking and Embedding) is a technology for communication between objects and embedding them within each other. The manual explains how OLE clients and servers can be implemented with the IDM.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Germany

Notation Conventions

DM will be used as a synonym for Dialog Manager.

The notion of UNIX in general comprises all supported UNIX derivates, otherwise it will be explicitly stated.

< >	to be substituted by the corresponding value
color	keyword
.bgc	attribute
{ }	optional (0 or once)
[ ]	optional (0 or n-times)
<A> | <B>	either <A> or <B>

Description Mode

All keywords are bold and underlined, e.g.

variable **integer** **function**

Indexing of Attributes

Syntax for indexed attributes:

[I]

[I,J] meaning [row, column]

Identifiers

Identifiers have to begin with an uppercase letter or an underline ('_'). The following characters may be uppercase or lowercase letters, digits, or underlines.

Hyphens ('-') are ***not*** permitted as characters for specifying identifiers.

The maximal length of an identifier is *31* characters.

Description of the permitted identifiers in the Backus-Naur form (BNF)

<identifier>	::=	<first character>{<character>}
<first character>	::=	_ | <uppercase>
<character>	::=	_ | <lowercase> | <uppercase> | <digit>

<digit> ::= 1 | 2 | 3 | … 9 | 0

<lowercase> ::= a | b | c | … x | y | z

<uppercase> ::= A | B | C | … X | Y | Z

Table of Contents

1 Introduction

This manual describes how to use the Dialog Manager for OLE communications. First the Dialog Manager is described as OLE client and then as OLE server. Both chapters contain examples illustrating the basic working methods with OLE.

The COM technology (Component Object Model) offers possibilities to have applications communicate with one another, including graphic integration. This technology is better known as OLE.

The project team may expand the application possibilities of existing programs by integrating new programs which have been defined with the Dialog Manager on the basis of standard interfaces (OLE). So, the Dialog Manager interfaces and features can be used in existing programs without that previous investments decrease in value.

1.1 Requirements

1.1.1 Developer

Using the OLE functionality requires working knowledge of the COM technology and Windows. You should also have experience with .reg files,.idl and .tlb files as well as with the meaning of interfaces in OLE and COM. Furthermore you should know the meaning of the terms "methods" and "properties" with regard to the Idispatch interface.

1.1.2 System

OLE functionality only works on the Microsoft Windows operating systems. The dialog itself, however, may run "standalone" on many other systems, on which the OLE server functionality described in the dialog has no meaning.

1.1.3 Container / Client

The application (in the following called OLE client or client) should access the interfaces of the OLE server via the IDispatch-Interface, in order to be able to use the methods and properties. Moreover there must be several interfaces available to allow the Dialog Manager to establish a communication with the client.

1.1.4 User

Users do not need special requirements, as they usually have no insight into the structure of the application as OLE client and OLE server. Due to the time needed for the communication a time delay

might occur when starting the server.

2 The control Object

The tasks of the control object depend on the way the control object is used. There are two possibilities:

- If the control object is used as OLE client, then the communication with the server will be established with this object. In addition, this object defines the area in which the server is to appear within the client. All calls made to the server to set and query properties or to call methods will thus be carried out via this control object.
- If the control object is used as OLE server, it is used to define the interface and to communicate with the client. Dialog Manager is not able to make all its objects, methods, resources and attributes, etc. available as interfaces. This would go beyond the scope of the interfaces in OLE and, in addition, it is not recommendable offering the entire dialog to the external program. The attributes, methods and events of the control object describe the interface of the OLE server to its clients.

Definition

```
{ export | reexport } { model } control { <Identifier> }
{
  <standard attributes>
  <plain attributes>
  <geometry attributes>
  <grid attributes>
  <hierarchy attributes>
  <layout attributes>
  <object-specific attributes>
}
```

Events

extevent

finish

help

help

paste

select

start

Children

canvas

checkbox

edittext

groupbox

image

listbox

menubox

menuitem

menusep

notebook

poptext

pushbutton

radiobutton

rectangle

scrollbar

spinbox

statictext

tablefield

treeview

window

Parent

dialog

groupbox

layoutbox

module

notepage

splitbox

toolbar

window

Menu

Pop-up menu

2.1 Attributes

Attributes	RSD	PID	Properties	Short Description
acc_label	string object	string text	S.G/D/C	overwrites the Automation Identifier for MICROSOFT UI Automation
acc_text	object string	text string	S.G/D/C	overwrites the Automation Name for MICROSOFT UI Automation
accelerator	identifier	accel	S,G/D/C	accelerator of object
active	boolean	active	S,G/D/C	state of server or client
bgc	identifier	color	S,G/D/C	background color of object
bordercolor	identifier	color	S,G/D/C	border color of object
borderwidth	integer	integer	S,G/D/C	width of object border
child[I]	object	object	S,G/-/C	accesses the I-th child object
childcount	integer	integer	-,G/-/-	queries the number of child objects
class	class	class	-,G/-/-	class of object
connect	boolean	boolean	S,G/D/C	state of connection to OLE server or OLE client
cursor	identifier	cursor	S,G/D/C	cursor belonging to object
cut_pending	boolean	boolean	S,G/-/-	cut operation is still pending
cut_pending_changed	boolean	boolean	-,G/-/-	changing state during cut operation
dialog	identifier	instance	-,G/-/-	dialog for object
document[I]	object	document	S,G/-/-	accesses the I-th XML Document
external	boolean	boolean	-,G/-/-	returns if the object class is an USW class

Attributes	RSD	PID	Properties	Short Description
external[I]	class	class	-,G/-/-	returns the I-th registered USW class
fgc	identifier	color	S,G/D/C	foreground color of object
firstchild	object	object	S,G/-/C	accesses the first child object
firstrecord	object	record	S,G/-/C	accesses the first record of an object
firstsubcontrol	object	object	S,G/D/C	accesses the first sub-control
focus	object boolean	instance boolean	-,G/-/-	keyboard focus of object
font	identifier	font	S,G/D/C	font of object
function	identifier	func	S,G/D/C	function of object
groupbox	identifier	instance	-,G/-/-	groupbox the object belongs to
height	integer	integer	S,G/D/C	indicates height of object
help	string identifier	string text	S,G/D/C	helptext of object
index	integer	integer	-,G/-/-	current index of object in the children list of its par-ent
label	string	string	S,G/D/C	name (identifier) of object
lastchild	object	object	S,G/-/C	accesses the last child object
lastrecord	object	record	S,G/-/C	accesses the last record of an object
lastsubcontrol	object	object	S,G/D/C	accesses the last sub-control
license_key	string	string	S,G/D/C	license key for an ActiveX control

Attributes	RSD	PID	Properties	Short Description
member[I]	attribute	attribute	-,G/-/-	i-th user-defined attribute of object
membercount	integer	integer	-,G/-/-	number of user-defined attributes
menu	identifier	instance	S,G/D/C	menu of object
message[I]	identifier	message	-,G/D/-	messages to be sent to client
mode	enum	enum	S,G/D/C	mode of control object, either as client or as server
model	identifier	instance	S,G/D/C	model belonging to object
name	string	string	-,G/D/-	name or ProgID of server or of client
notepage	identifier	instance	-,G/-/-	notepage the object belongs to
parent	identifier	instance	S,G/-/-	parent of object
picture	identifier	instance	S,G/D/C	picture to be displayed in inactive state
posraster	boolean	boolean	S,G/D/C	indication of position refers to raster
real_height	integer	integer	-,G/-/-	real height of object
real_sensitive	boolean	boolean	-,G/-/-	real selectability of object
real_visible	boolean	boolean	-,G/-/-	real visibility of object
real_width	integer	integer	-,G/-/-	real width of object
real_x	integer	integer	-,G/-/-	real distance from the left (in pixel)
real_xraster	integer	integer	-,G/-/-	width of internally used raster
real_y	integer	integer	-,G/-/-	real distance from top (in pixel)

Attributes	RSD	PID	Properties	Short Description
real_yraster	integer	integer	-,G/-/-	height of internally used raster
record[I]	object	record	S,G/-/C	accesses the I-th record of an object
recordcount	integer	integer	-,G/-/-	queries the number of child records
reffont	identifier	font	S,G/D/C	reference font of object
scope	integer (1, 2, 3)	scope	-,G/-/-	queries the object type (Default, Model or instance)
sensitive	boolean	boolean	S,G/D/C	selectability of object
sizeraster	boolean	boolean	S,G/D/C	size refers to raster of par-ent object
statushelp	string identifier	string text	S,G/D/C	text to be displayed in the statusbar
subcontrol[I]	object	object	S,G/D/C	accesses the I-th sub-control
subcontrolcount	integer	integer	-,G/-/-	queries the number of sub-controls
toolhelp	string object	string text	S,G/D/C	gives a short explanation of object at the cursor
userdata	anyvalue	anyvalue	S,G/D/C	userdata of object of any datatype
uuid	string	string	-,G/D/-	unambiguous UUID of OLE server
visible	boolean	boolean	S,G/D/C	visibility of object
width	integer	integer	S,G/D/C	current width of object
window	identifier	instance	-,G/-/-	window the object belongs to
xauto	integer (-1, 0, 1)	integer	S,G/D/C	type of x-coordinates defin-ition

Attributes	RSD	PID	Properties	Short Description
xleft	integer	integer	S,G/D/C	x-coordinate, distance from the left
xraster	integer	integer	S,G/D/C	unit in x direction
xright	integer	integer	S,G/D/C	x-coordinate, distance from the right
yauto	integer (-1, 0, 1)	integer	S,G/D/C	type of y-coordinates definition
ybottom	integer	integer	S,G/D/C	y-coordinate, distance from bottom
yraster	integer	integer	S,G/D/C	unit in y direction
ytop	integer	integer	S,G/D/C	y-coordinate, distance from top

2.2 Specific Attributes

2.2.1 connect

Identifier:

.connect

Classification: object-specific attribute

Definition

argument type: boolean

C definition: AT_connect

C datatype: DT_boolean

COBOL definition: AT-connect

COBOL datatype: DT-boolean

access: set, get

“changed”, i.e. attribute can be used to trigger rules.

This attribute defines the state of the connection to the OLE client or to the OLE server. The services of the OLE server can only be used if there is a connection.

2.2.2 mode

Identifier:

.mode

Classification:	object-specific attribute

Definition

argument type:	enum
value range:	mode_none, mode_client, mode_server
C definition:	AT_mode
C datatype:	DT_enum
COBOL definition:	AT-mode
COBOL datatype:	DT-enum
access:	set, get

“changed”, i.e. attribute can be used to trigger rules.

This attribute defines how to use the control object. There are three possibilities:

- mode_none: The mode is not defined, i.e. control is not used, neither as client nor as server. For example, you can set this value if you have implemented an OLE server, but if, for some reason, you are not able to act as a server, e.g. because the program was started as a normal program.
- mode_client: The control object will be regarded as OLE client, i.e. an OLE server will be accessed via this object.
- mode_server: The control object is used as OLE server and may be accessed by other clients.

2.2.3 name

Identifier:

.name

Classification:	object-specific attribute

Definition

argument type:	string
C definition:	AT_name

C datatype:	DT_string
COBOL definition:	AT-name
COBOL datatype:	DT-string
access:	set, get

“changed”, i.e. attribute can be used to trigger rules.

This attribute defines the name of the corresponding server at the control object which is used as OLE client. You may deposit either the name or the ProgID of the server in this attribute.

If the control object is used as OLE server, the given string in this attribute will be registered as server name in the registry list.

Example

```
model control CtTest
{
  .mode mode_client;
  .name "InternetExplorer.Application.1";
  .visible true;
  .active true;
  .connect false;
}
```

2.2.4 picture

Identifier:

.picture

Classification:	object-specific attribute

Definition

argument type:	object
C definition:	AT_picture
C datatype:	DT_tile
COBOL definition:	AT-picture
COBOL datatype:	DT-tile
access:	set, get

“changed”, i.e. attribute can be used to trigger rules.

This attribute is used to define the picture which is to be displayed in the inactive state of the control object.

See Also

Attribute picture

2.2.5 uuid

Identifier:

.uuid

Classification:	object-specific attribute

Definition

argument type:	string
C definition:	AT_uuid
C datatype:	DT_string
COBOL definition:	AT-uuid
COBOL datatype:	DT-string
access:	get

This attribute must be set when the control object is used as OLE server. Via this UUID the object can be clearly identified by other programs. This UUID will be generated by using the program guidgen.exe.

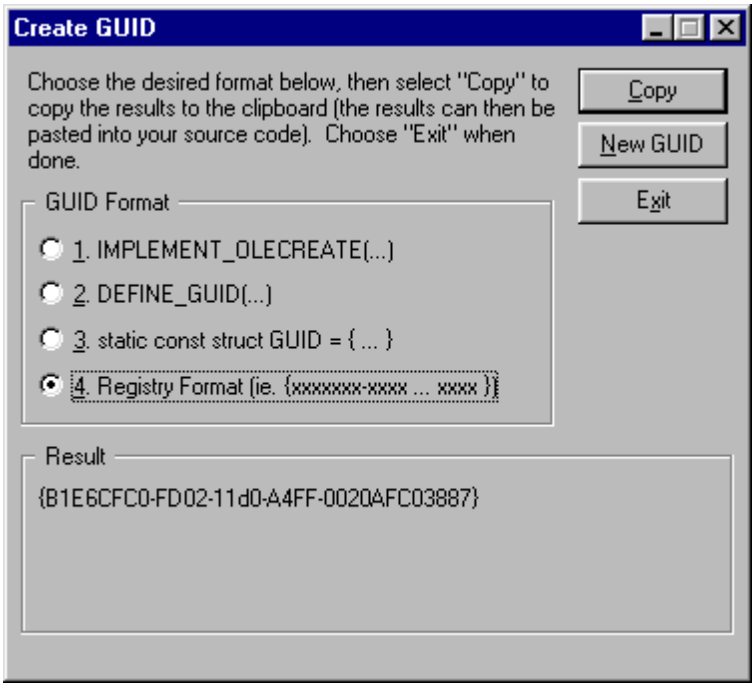

Figure 1: *Generation of GUID*

If the control object is used as OLE client, you may deposit the server UUID in this attribute. This attribute thus replaces the attribute .name which is usually used to deposit the server name.

3 The subcontrol Object

OLE objects that can be used as servers by the IDM through the IDM object ***control*** often have a rather complex structure and can consist of several further child objects. For example, the OLE control “Word.Application” has a collection of “Documents”, which manages the open documents. In turn, these documents have “Words”, “Sentences”, “Ranges”, etc. as their own children. To be able to access these sub-objects from the Rule Language, i.e. to call methods or query attributes, the ***subcontrol*** object can be used. The ***subcontrol*** represents an OLE child object which can be directly or indirectly managed by an OLE server. Therefore the starting point for accessing such an object is always the ***control***.

Definition

```
{ export | reexport } subcontrol { <Identifier> }
{
  <hierarchy attributes>
  <object-specific attributes>
}
```

Events

None

Children

document

record

subcontrol

transformer

Parents

control

subcontrol

Menu

None

3.1 Attributes

connect

control

dialog

document[I]

firstrecord

firstsubcontrol

groupbox

label

lastrecord

lastsubcontrol

layoutbox

model

module

notepage

parent

record[I]

recordcount

scope

subcontrol[I]

subcontrolcount

toolbar

transformer[I]

userdata

window

3.2 Implicit Creation

One notable feature of subcontrols is that these objects can be implicitly created without being statically defined as object in a dialog or being dynamically created with **create()** before. But please pay attention to chapter “Dynamic OLE Properties and Subcontrols” when you work with dynamic OLE properties which create subcontrols.

Probably the implicit use is the most common use of these objects in practice. For example, when using Microsoft Word as an OLE server the individual can be accessed like this:

Let ***control***“Co” be defined somewhere in the dialog.

```
child control Co
{
  .mode mode_client;
  .name "Word.Application";
  .visible true;
```

```
  .connect true;
}
```

Then control Co can be used in a rule like this:

```
rule OLETest ()
{
  variable object Doc;
  Doc := Co.Documents:Add();
  // Word has a collection that is entitled "Documents".
  // Co.Documents is used to access this OLE object. In order to
  // address the object in IDM, a subcontrol with the identifier
  // "Documents" is implicitly created and registered as a child
  // of control Co. The :Add() method is a member of the
  // "Documents" collection. Therefor, the method can be applied
  // to the just created subcontrol. It creates a new document
  // in Word and returns a pointer to its IDispatch interface.
  // As a counterpart on the IDM side another subcontrol is
  // created as child of the first subcontrol and connected
  // with the Word document. Finally this object is assigned
  // to the variable "Doc".

  // The subcontrol in "Doc" now can be used to call methods of
  // the OLE document and to query or set its properties.
  print Doc.FullName;
}
```

In general, if a method or a property returns a pointer to an IDispatch interface, or if this is passed to the IDM as a parameter, the IDM creates a subcontrol that is connected to the IDispatch interface. This means that the *.connect* attribute of such a subcontrol is already set to *true*.

Normally the identifier (label) of such a created object is not set so that something similar to *"sub-control Co.SUBCONTROL[2]"* appears in the trace file. One exception is the access of properties that return OLE objects. Since these are usually collections whose names can be known to the IDM, an implicitly created subcontrol receives the name of the property as identifier.

Example

```
Doc1 := Co.Documents:Add();
Doc2 := Co.Documents:Item(1); // Assumption: No other Word documents
                             // were open before.
```

In this example, two subcontrols are created in the first line: One with the identifier "Documents", another with the identifier "SUBCONTROL". The reason for the identifier of the second subcontrol is that the IDM has no further information when processing the Add() method. In the second line only one more subcontrol is created with "SUBCONTROL[2]" as its identifier. As the IDM recognizes that a subcontrol for "Documents" already exists, it is unnecessary to create it once more. All of this leads to the situation, that on the OLE side there are two objects: one is the "Documents" collection and the other one an empty document. On the IDM side there are three objects: a subcontrol for the

"Documents" collection and two subcontrols in the variables "Doc1" and "Doc2", both connected to the empty document on the OLE side. It is a matter of good programming style to avoid heaps of unnecessary subcontrol objects.

One may wonder at this point, what happens to all these subcontrols if they are not needed anymore. Chapter "Garbage Collection" deals with this.

3.3 Dynamic OLE Properties and Subcontrols

When dynamic properties of OLE objects are queried, this may lead to the creation of new subcontrols when these properties return objects (e.g. "ActiveSheet" of MICROSOFT EXCEL). At first this is no problem. In subsequent queries however, the IDM will not refer to a current, newly returned object but to the already existing subcontrol which has been cached by the IDM.

If this caching is unwanted or leads to problems, it can be avoided by two means:

- After the access of a OLE property that creates a subcontrol, this subcontrol can be destroyed with the **:destroy()** method before the next access of the OLE property (provided that it is no longer needed).
- After the access of a OLE property that creates a subcontrol, the *.label* attribute of this subcontrol can be set to an empty string (""). Hereby the subcontrol object remains connected and available. It will be destroyed within the usual garbage collection (see chapter "Garbage Collection"). Further accesses of the OLE property will create a new subcontrol as the prior object will not be found (internal search relies on *.label*).

Setting *.connect* of the concerned subcontrol to *false* provides no solution however, because with this the subcontrol persists, only all further accesses will fail.

3.4 Garbage Collection

All implicitly created subcontrols have to be deleted again. For this purpose, the IDM remembers by how many objects a subcontrol is referenced. If it is determined that a subcontrol is no longer in use, neither by a variable (local, global, static) nor by a user-defined attribute, then the subcontrol is destroyed. Currently two strategies are applied, which are explained below.

1. If an implicitly created subcontrol is assigned to a variable (or something similar), and later this variable is overwritten with another value, the subcontrol can be released, provided that it has no children.

   Example

```
Doc := Co.Documents:Add();  // 2 Subcontrols are implicitly created (A for
Documents
                            // and B for the document created by the Add()
method).
Doc := Co;                  // Subcontrol B is no longer used by the
variable Doc
```

```
                                // and therefore can be destroyed. Now
subcontrol A can
                                // determine that it has no more children and
can destroy
                                // itself too.
```

2. In awkward situations it may happen that a subcontrol does not notice it can be destroyed. This happens for example, when a newly created subcontrol has not been assigned anywhere. This situation lacks a trigger to throw away the object. For this reason the IDM occasionally starts a cleanup in which implicitly created subcontrols are checked if they are still required. Otherwise they are discarded.

Important

Since the IDM garbage collection only takes into consideration the references from the Rule Language, great care should be taken, if ObjectIDs of subcontrols are managed from the C interface (as well as the COBOL and C++ interfaces). If such an ID is temporarily stored in the application, this is not realized by the IDM. After the call of interface functions, the IDM may determine that the subcontrol is not needed anymore and dispose it accordingly. If control is transferred back to the application side, it can no longer be assumed that the previously saved ObjectID is still valid. Therefore it should be avoided to save subcontrols in the application. If this is wanted still, it is important to ensure that the object is still referenced in the Rule Language (for example by a global or static variable, or in a user-defined attribute). This guarantees that the subcontrol will not be thrown away.

3.4.1 Example

The following example shows how subcontrols can be used.

```
dialog Word

window WnWindow
{
  .active false;
  .width 400;
  .height 100;
  .title "Word.Document.8";

  on close
  {
    if Co.connect then
      Co:Quit();
    endif
    exit();
  }

  child control Co
  {
    .visible true;
```

```
  .active false;
  .xauto 0;
  .xleft 20;
  .xright 20;
  .yauto 0;
  .ytop 20;
  .ybottom 40;
  .mode mode_client;
  .name "Word.Application";
  .connect true;
}

child pushbutton PbOpen
{
  .xauto 1;
  .xleft 100;
  .width 76;
  .yauto -1;
  .height 27;
  .ybottom 10;
  .text "Open Doc";
  on select
  {
    variable object Doc:=null;

    // Assumes that the file actually exists
    Doc := Co.Documents:Open("c:/tmp/olddocument.docx");

    if Doc <> null then
      print "DocName: " + Doc.FullName;
      print "DocSaveStatus: " + Doc.Saved;
    endif
  }
}

child pushbutton PbAdd
{
  .xauto 1;
  .xleft 20;
  .width 76;
  .yauto -1;
  .height 27;
  .ybottom 10;
  .text "New Doc";
  on select
  {
```

```
      // Assumes that the file actually exists
      Co.Documents:Add("c:/tmp/newdocument.docx");
    }
  }

  child pushbutton PbClose
  {
    .xleft 180;
    .width 92;
    .yauto -1;
    .height 27;
    .ybottom 10;
    .text "Close Doc";
    on select
    {
      // Close first document (of the document list)
      Co.Documents:Item(1):Close();
    }
  }

  child pushbutton PbQuit
  {
    .xauto -1;
    .width 85;
    .xright 20;
    .yauto -1;
    .height 27;
    .ybottom 10;
    .text "Quit";
    on select
    {
      // Close Word
      Co:Quit();
    }
 }
}

on dialog start
{
  Co.Visible := true;
}
```

4 Dialog Manager as OLE Client

This chapter describes how to integrate OLE objects into Dialog Manager.

In the client mode, the control object establishes a connection to the OLE server. With the following attributes you define which server is to appear, as well as the time and the form of its appearance.

- .picture defines the displayed picture in the inactive state
- .name defines the name of the external server
- .connect defines the connection state
- .visible defines the visibility
- .active defines the activation state
- .xleft, .ytop, ... defines the geometry

4.1 Server Integration

The server is actually integrated into the dialog via the Control object. The following attributes must contain the corresponding values:

- .name must contain the name or the ProgID of the used server
- .xleft, .ytop, ... defines the position at which the server is to appear, if the Control object has not been defined on top level
- .visible defines whether Control is to be visible
- .sensitive defines whether Control is to be selectable
- .active defines whether server is to be active and thus accessible
- .connect indicates whether there is a connection to the server
- .picture contains the picture to be displayed in the inactive state of the server

Example

```
tile Server_Picture "IDM_IMAGES:isaicon.gif";
window WnClient
{
  .title "Integrate OCX in IDM dialog";
  child control Ctrl_Grid
  {
    .visible true;
    .active false;
    .xleft 11;
    .width 249;
    .ytop 48;
    .height 169;
    .picture Server_Picture;
```

```
    .mode mode_client;
    .name "MSGrid.Grid";
    .connect false;
  }
}
```

4.2 Activation of Server

The actual server activation is carried out by setting the attributes

- .connect
- .active

to value TRUE. If both attributes contain this value, there is a connection to the OLE server; i.e. methods may be called and/or attributes may be called and set.

You should always check whether the attribute .connect is set to TRUE, as it might occur that the server cannot be started.

Example

```
rule void activation
{
  setvalue(Ctrl_Grid, .connect, true, true);
  !! this must be checked here
  !! as the connection
  !! might fail
  if (Ctrl_Grid.connect = true) then
    WnClient.title := "Client + Server";
    !! finally the client is made active
    Ctrl_Grid.active := true;
  else
    print "could not connect";
  endif
}
```

4.3 Using the Server

Calling methods as well as querying and setting attributes (properties) at the server is carried out exclusively via the control object. For this purpose a syntax corresponding to the user-defined attributes and methods is used. These attributes and methods, however, must not be defined at the control object. If a user-defined attribute is accessed at the control object and if this attribute is not defined there, it will be passed on to the server. The same applies to the methods. If a method is called at the control object and if this method is not defined there, this method will be called at the server. If it is not defined there either, an error occurs, which will be traced in the tracefile or logfile.

Example

The attributes .Cols, .Rows and :GridLines are attributes of the used OLE server and may thus not be defined in the control object.

```
child control Ctrl_Grid
{
    .visible true;
    .active false;
    .xleft 11;
    .width 249;
    .ytop 48;
    .height 169;
    .picture Server_Picture;
    .mode mode_client;
    .name "MSGrid.Grid";
    .connect false;

    rule void activation
    {
    variable integer Cols := 0;
    variable integer Rows := 0;

    setvalue(Ctrl_Grid, .connect, true, true);
    !! this must be checked here
    !! as the connection
    !! might fail
    if (Ctrl_Grid.connect = true) then
        WnClient.title := "Client + Server";
        Cols := atoi(EtCols.content);
        Rows := atoi(EtRows.content);
        !! set the specified lines and the given number of
        !! columns
        if (Cols <> 0) then
        Ctrl_Grid.Cols := Cols;
        endif
        if (Rows <> 0) then
        Ctrl_Grid.Rows := Rows;
        endif
        Ctrl_Grid.GridLines := CbGridLines.active;
        !! finally the client is made visible
        Ctrl_Grid.active := true;
    else
        print "could not connect";
    endif
   }
```

4.4 Identifying the Interfaces

If you want to use an OLE server you have to know the interfaces. These interfaces consist of attributes (properties) and methods. There are several possibilities to get to know these interfaces:

- reading the manual: For the OLE servers you use you have to refer to the corresponding manual in which the attributes and methods are clearly defined. This is the best way, but such manuals are often difficult to find.
- checking with Microsoft C++ compilers: From version 4.0 on, the Microsoft C++ compiler offers means to query interfaces of OLE objects.

4.4.1 Microsoft C++ Compiler Version 4.x

To get information about OLE objects you have to proceed as follows with version 4 of the Microsoft C compiler:

- start OLE-COM Object Viewer via the Developer Studio (menu "Tools")
- select relevant control
- select relevant interfaces (usually IDispatch....) with a double click (in the displayed list only the bold interfaces are really available)
- select relevant information in the combobox on the top left in the new window
- select relevant method in the upper left listbox or the relevant attributes in the upper right listbox to get more information (datatype, parameter, ...).

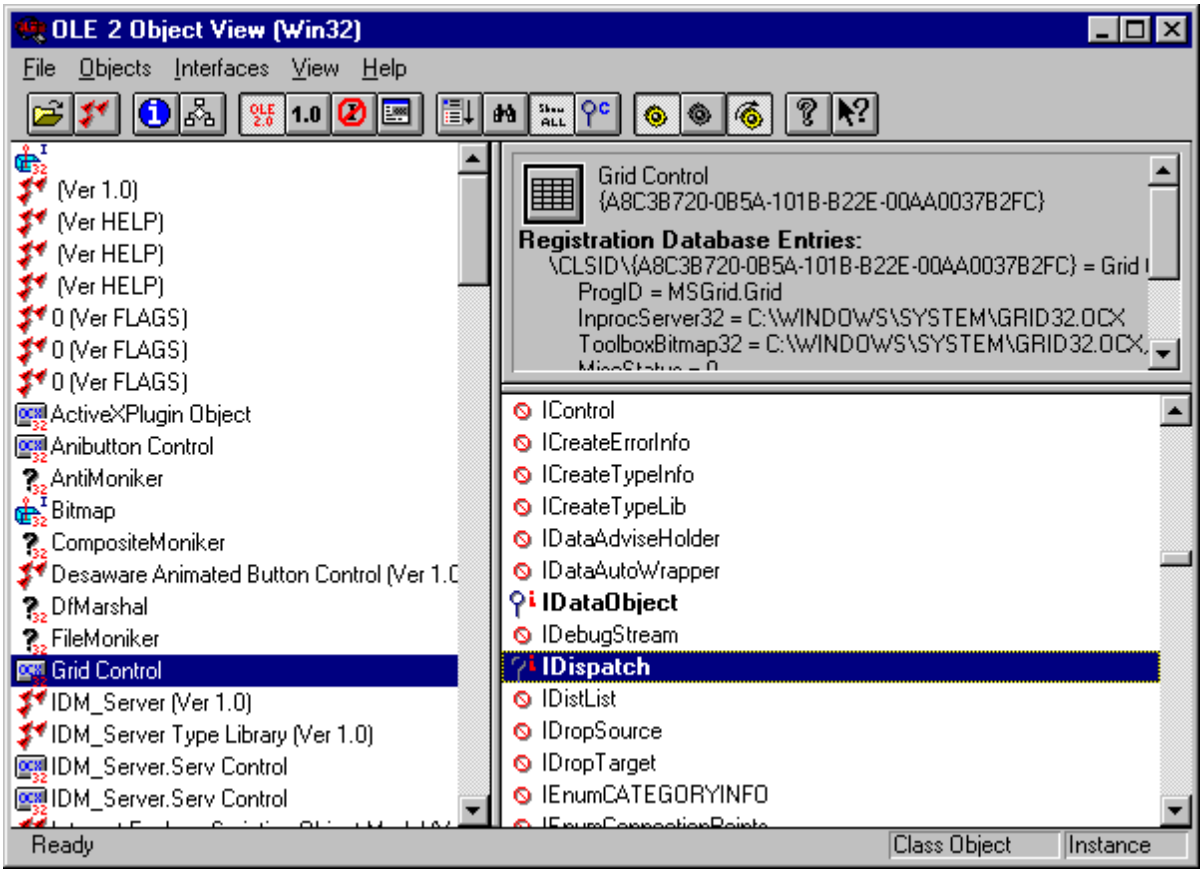

Figure 2: *OLE2 Object View of MSC version 4.0 after the start*

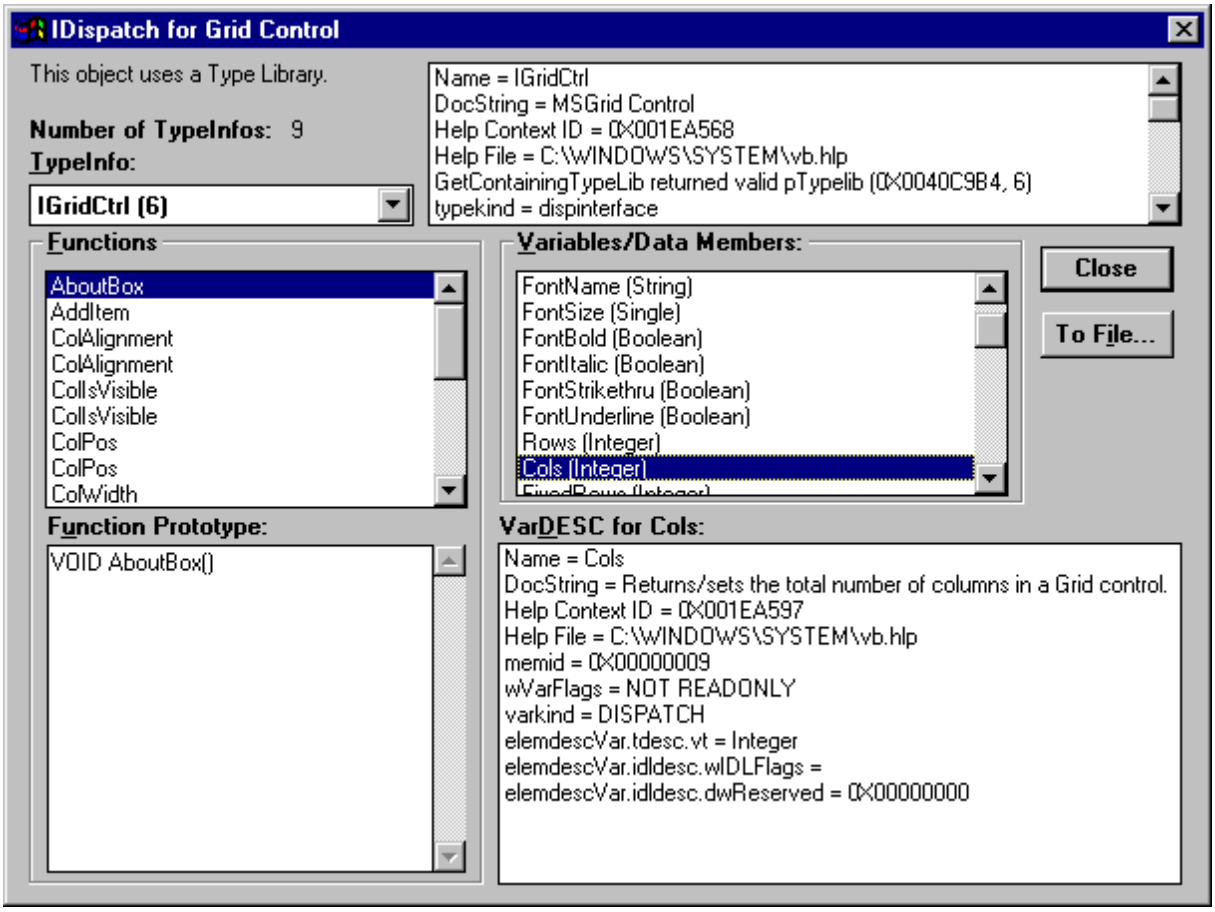

Figure 3: *Selection of relevant attribute in MSC version 4.0*

4.4.2 Microsoft C++ Compiler Version 5.0

To get information about OLE objects you have to proceed as follows with version 5 of the Microsoft C compiler:

- start OLE-COM Object Viewer
- select item "Control" in the left list
- select relevant control
- select menuitem "View Type Information..." in menu "Object" or in popup menu
- select relevant interfaces (usually dispinterface ....)
- select item "Methods" to view methods, or "Properties" to query attributes
- select relevant method or relevant attribute to get more information (datatype, parameter, ..)

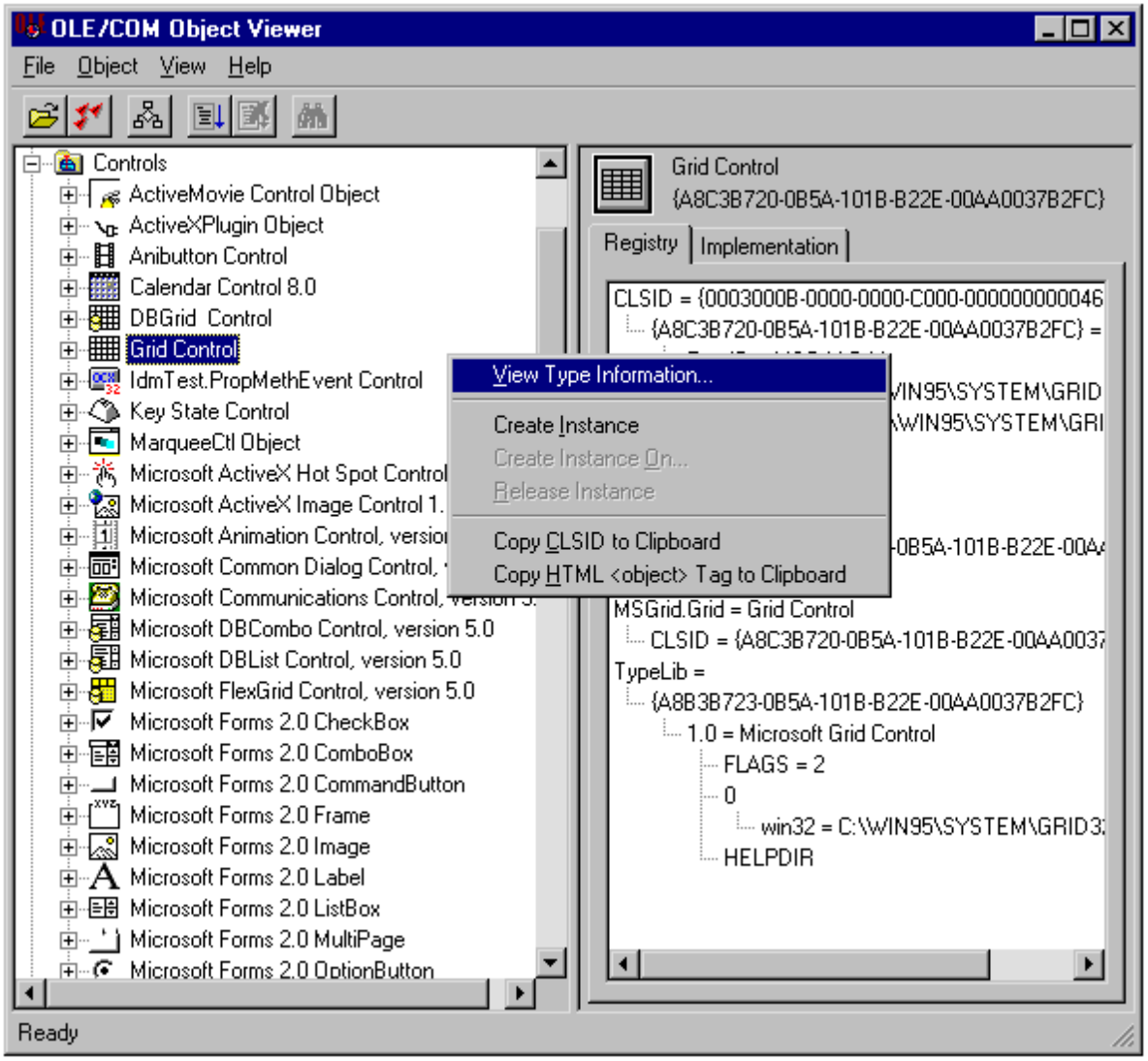

Figure 4: *Select control in MSC version 5.0*

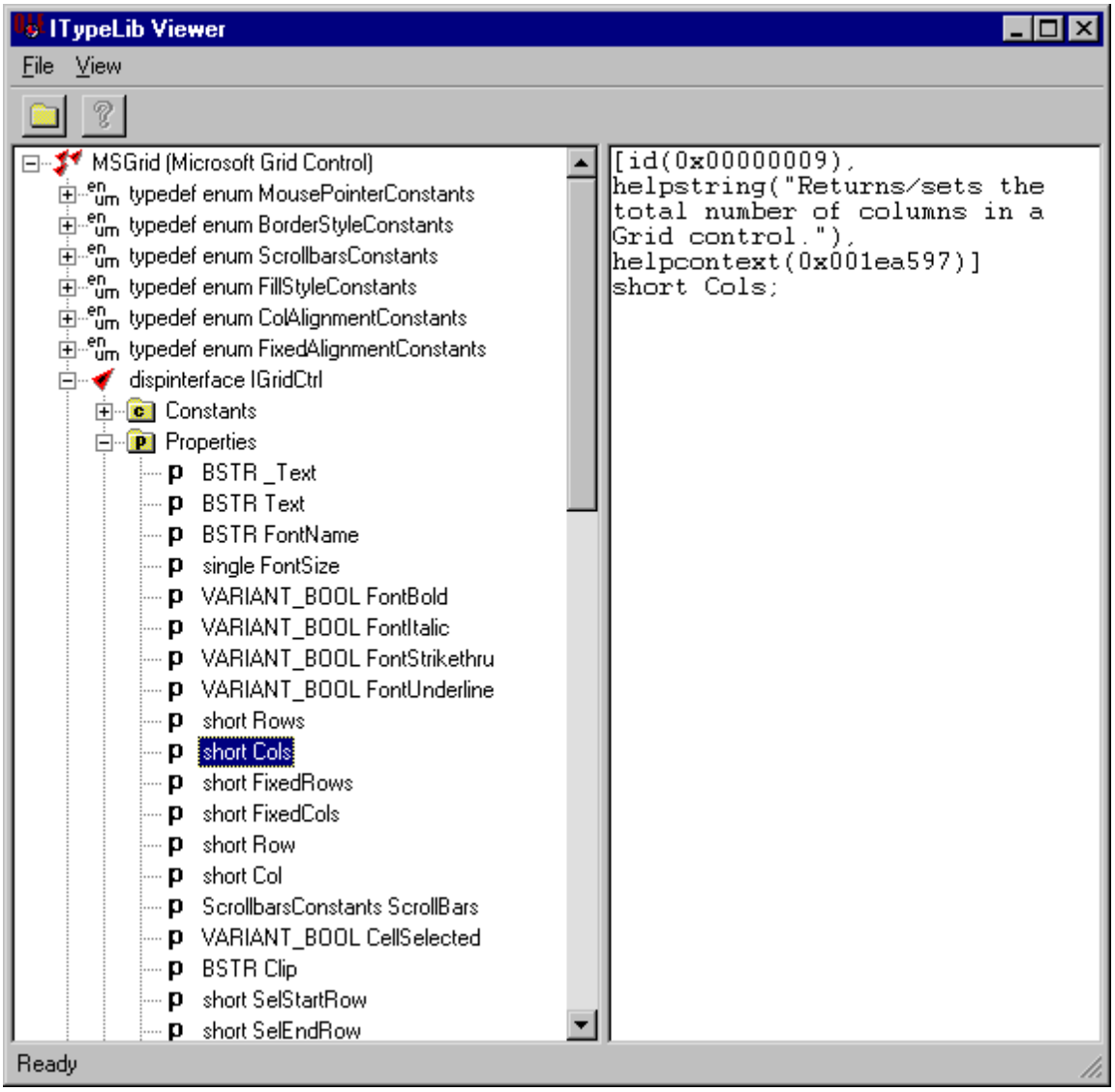

Figure 5: *Select a property in MSC version 5.0*

4.5 Use of Grid Controls

This example shows how to integrate and use a grid control in a dialog.

```
dialog Ocx
{
}
color CLRed rgb(255,0,0), grey(0);
font FnInv "10.MS Serif";
tile Server_Picture "IDM_IMAGES:isaicon.gif";
model pushbutton MpbVisible
{
  .xleft 458;
  .ytop 157;
  .text "Visible";
```

```
  boolean Visible := false;
}
window WnClient
{
  .active false;
  .xleft 87;
  .width 560;
  .ytop 132;
  .height 233;
  .iconic false;
  .title "Integration of OCX in IDM dialog";
  on close
  {
      Ctrl_Grid.connect := false;
      exit();
  }
  child control Ctrl_Grid
  {
    .visible true;
    .active false;
    .xleft 11;
    .width 249;
    .ytop 48;
    .height 169;
    .picture Server_Picture;
    .mode mode_client;
    .name "MSGrid.Grid";
    .connect false;
    rule void activation
    {
    variable integer Cols := 0;
    variable integer Rows := 0;

    setvalue(Ctrl_Grid, .connect, true, true);
    !! this must be checked here
!! as connection
    !! might fail
    if (Ctrl_Grid.connect = true) then
        WnClient.title := "Client + Server";
        Cols := atoi(EtCols.content);
        Rows := atoi(EtRows.content);
        !! set specified lines and given number of columns
        if (Cols <> 0) then
        Ctrl_Grid.Cols := Cols;
        endif
        if (Rows <> 0) then
```

```
      Ctrl_Grid.Rows := Rows;
      endif
      Ctrl_Grid.GridLines := CbGridLines.active;
      !! finally the client is made visible
      Ctrl_Grid.active := true;
  else
      print "could not connect";
  endif
  }
}
child pushbutton PbStart
{
  .xleft 282;
  .width 90;
  .ytop 32;
  .height 50;
  .text "Start Server";
  on select
  {
  Ctrl_Grid:activation();
  }
}
child rectangle
{
  .xleft 265;
  .width 6;
  .ytop 4;
  .height 369;
}
child statictext
{
  .sensitive false;
  .xleft 276;
  .width 204;
  .ytop 10;
  .text "Client";
}
child statictext
{
  .sensitive false;
  .xleft 7;
  .width 206;
  .ytop 10;
  .height 18;
  .text "Server area";
}
```

```
child pushbutton PbStop
{
  .xleft 413;
  .width 90;
  .ytop 30;
  .height 50;
  .text "Stop Server";
  on select
  {
  Ctrl_Grid.connect := false;
  }
}
child checkbox CbGridLines
{
  .xleft 286;
  .ytop 176;
  .text "Grid lines";
  .state state_checked;
  on select
  {
  Ctrl_Grid.GridLines := this.active;
  }
}
child statictext StCols
{
  .sensitive false;
  .xleft 281;
  .ytop 139;
  .text "Columns:";
}
child statictext StRows
{
  .sensitive false;
  .xleft 282;
  .ytop 100;
  .text "Lines:";
}
child spinbox SpCols
{
  .visible true;
  .xleft 345;
  .width 60;
  .ytop 133;
  .height 25;
  .curvalue 2;
  on scroll
```

```
  {
  variable integer Cols := 0;

  Cols := atoi(this.EtCols.content);
  if (Cols <> 0) then
      Ctrl_Grid.Cols := Cols;
  endif
  }
  child edittext EtCols
  {
    .active false;
    .xauto 0;
    .xleft -1;
    .xright 1;
    .yauto 0;
    .ytop 0;
    .ybottom 0;
    .maxchars 2;
    .content "2";
    .multiline false;
    .startsel 0;
    .endsel 1;
    on charinput
    {
    variable integer Cols := 0;

    Cols := atoi(this.content);
    if (Cols <> 0) then
        Ctrl_Grid.Cols := Cols;
    endif
    }
  }
}
child spinbox SpRows
{
  .xleft 345;
  .width 60;
  .ytop 97;
  .curvalue 4;
  on scroll
  {
  variable integer Rows := 0;

  Rows := atoi(this.EtRows.content);
  if (Rows <> 0) then
      Ctrl_Grid.Rows := Rows;
```

```
    endif
    }
    child edittext EtRows
    {
      .active false;
      .xauto 0;
      .xleft -1;
      .xright 1;
      .yauto 0;
      .ytop -1;
      .ybottom 1;
      .maxchars 2;
      .content "4";
      .multiline false;
      .startsel 0;
      .endsel 1;
      on charinput
      {
      variable integer Rows := 0;

      Rows := atoi(this.content);
      if (Rows <> 0) then
          Ctrl_Grid.Rows := Rows;
      endif
      }
    }
  }
}
```

When starting this dialog you get the following picture:

Figure 6: *Example of grid control after the start*

After selecting the "Start Server" pushbutton the OLE server is started and activated. The OLE server will then be displayed instead of the picture in the client.

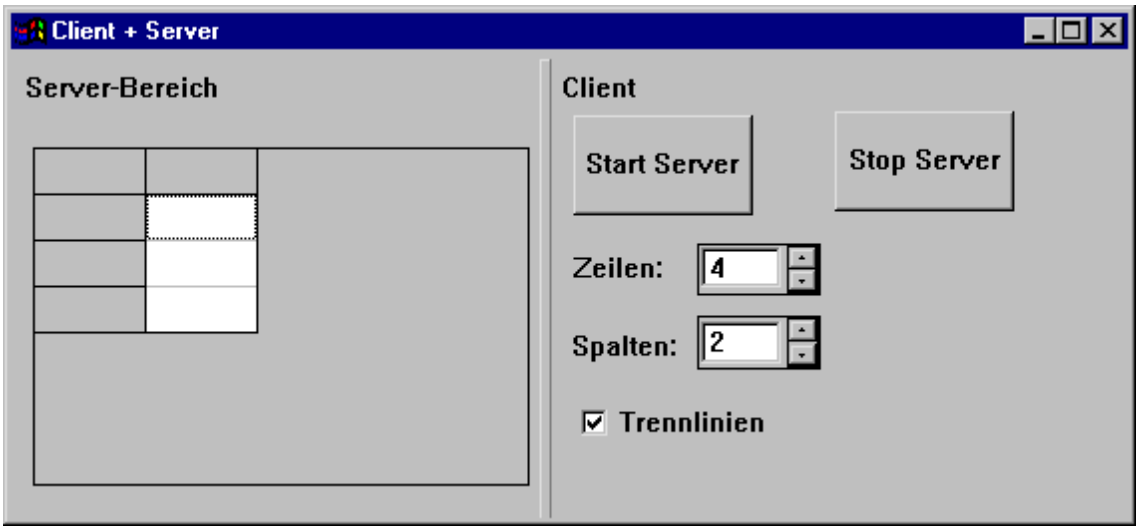

Figure 7: *Active grid control*

The attributes of the server can now be modified by using the spinboxes.

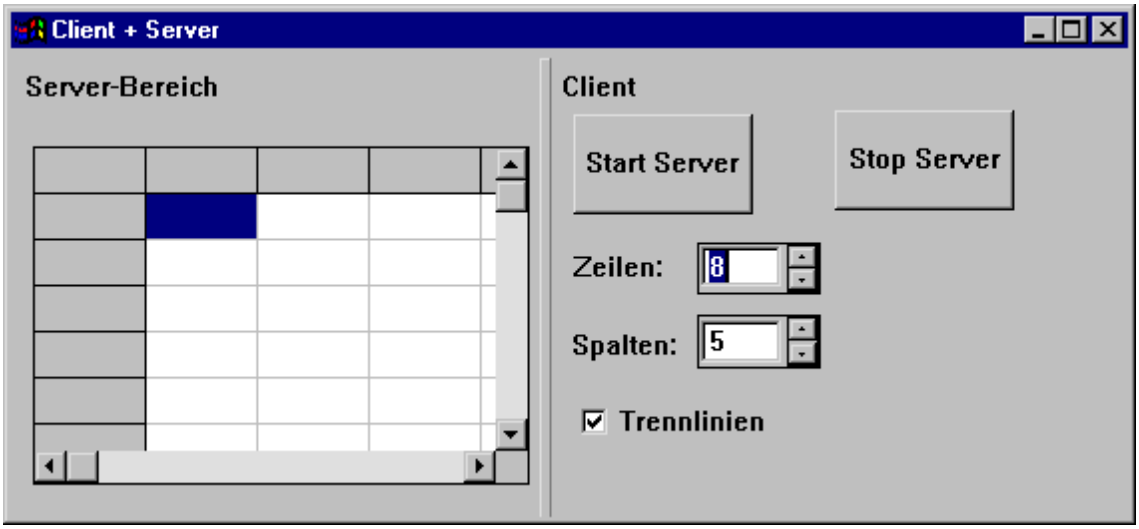

Figure 8: *Grid control with modified attributes*

4.6 Using the Internet Explorer

This example illustrates how to control the Internet Explorer from a Dialog Manager application.

```
dialog Client2
{
}
model control CtTest
{
    .mode mode_client;
    .name "InternetExplorer.Application.1";
    .visible true;
    .active true;
    .connect false;
}
window Window1
{
  .title "Internet Explorer Remote control";
  .width 400;
  .height 303;
  object Ctrl := null;

  child pushbutton Pb_Start
  {
    .xleft 37;
    .ytop 43;
    .text "Start";

    on select {
        if( Window1.Ctrl = null) then
```

```
      !! now a control is generated which the server
      !! is to include.
         Window1.Ctrl := create(CtTest, this.dialog, true);
      endif
      !! a connection is established
      !! the OLE server is set to visible
      !! now the server can be accessed, however
      !! it is not really visible
      Window1.Ctrl.connect := true;
      Window1.Ctrl.visible := true;
      if(Window1.Ctrl.connect <> true) then
          print "ERROR";
          print "Window1.Ctrl: " + Window1.Ctrl;
      endif
  }
}
child pushbutton Pb_Quit
{
  .xleft 143;
  .ytop 41;
  .text "Quit";
  on select {
      !! quit Internet Explorer
      !! by calling the method
      Window1.Ctrl:Quit();
      !! destroy the control object
      destroy(Window1.Ctrl);
  }
}
child pushbutton Pb_visible
{
  .xleft 34;
  .ytop 126;
  .text "Visible";

  on select {
      !! setting visible th OLE server
      Window1.Ctrl.Visible := true;
  }
}
child pushbutton Pb_navigate
{
  .xleft 29;
  .ytop 193;
  .text "Navigate";
  on select{
```

```
        !! Remote control of Internet Explorer
        Window1.Ctrl:Navigate(Et1.content, nothing,
            nothing, nothing, nothing);
    }
  }
  child edittext Et1
  {
    .xleft 143;
    .width 175;
    .ytop 192;
  }
}
```

After starting the application the following start window appears:

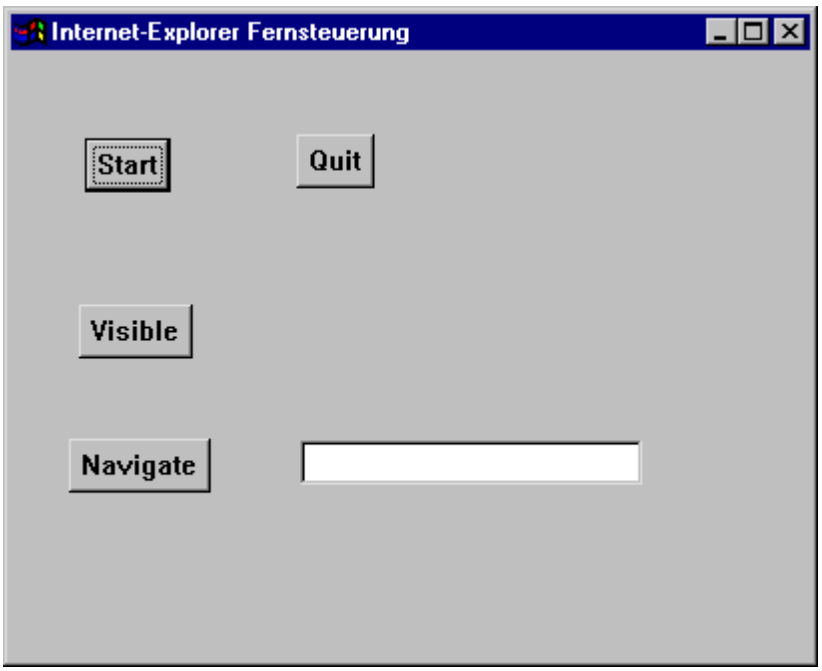

Figure 9: *Window for Internet Explorer control*

After selecting the "Start" pushbutton and the "Visible" pushbutton the Internet Explorer appears on the screen:

Figure 10: *Started Internet Explorer*

After entering an Internet location, here "www.sdr3.de", and after selecting the "Navigate" pushbutton the Internet Explorer displays the corresponding site.

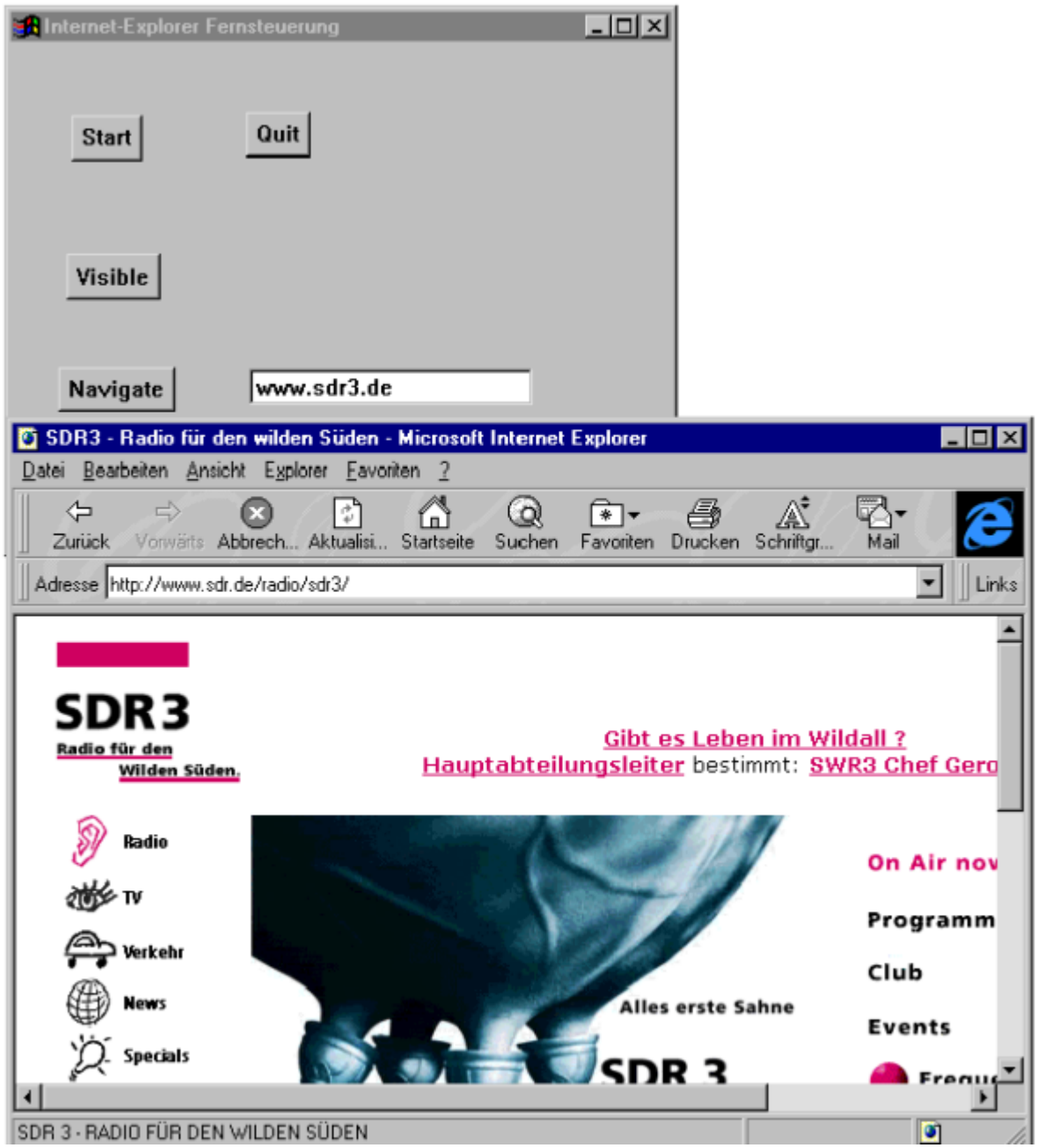

Figure 11: *Internet Explorer after executing the navigation*

5 Dialog Manager as OLE Server

The Dialog Manager cannot make all its objects, methods, resources, attributes, etc. available as interfaces. This would go beyond the scope of the interfaces in OLE and, in addition, it is not recommendable offering the entire dialog to the external program. The communication interface made available by Dialog Manager has the form of an object. This object belongs to the control class . The attributes, methods and events of the control object describe the interface of the OLE server to its clients.

5.1 Basic Structure of Control as OLE Server

The control class describes the interface to the client. Control may be instantiated several times, i.e. it may be used several times as different instances in one and the same application. This is why the control object may be exported as model. However, it is also possible to permit only one instance; in this case you have to declare the control as normal instance and not as model. Exported control objects may be defined only in the dialog itself and not in the modules to be reloaded, as available controls must be registered immediately on application start. This is why the control object must not be modified, e.g. by generating new attributes for this object.

Control has user-defined attributes and methods which correspond to the properties and methods in the usual OLE language. Control can have exactly one child. This child is the object to appear in the client. Apart from this child the control can have record objects which are not exported.

To be able to use a control object and its dialog as OLE server, the value mode_server must be assigned to the attribute .mode.

Attributes describe the properties. It is always possible to set or to get these attributes. Valid attributes are scalar, vector, and associative attributes with the data types and index types boolean, string and integer. Shadowing other user-defined or predefined attributes is also valid.

Methods describe the methods of the interface. These methods can be used by the application. The possible data types are restricted with regard to some parameters and return values: valid are boolean, string and integer.

Control has the attribute .picture. The picture will be displayed in the client when the server is inactive, provided that the client is able to do so. If no .picture is available, nothing will be displayed in the client. If the client is to permit activation and deactivation, then the child of control will be displayed in the client, as long as control is InPlace-active.

5.2 Unambiguous Interface

The client needs an unambiguous interface to be able to access the OLE server. This interface usually depends on the program and its version, as the client must always be able to identify the program and its interface unambiguously. A 128-bit code at the OLE guarantees the non-ambiguity of the interface. This 128-bit code is called UUID.

These UUIDs are given

» at the dialog and

» at each control used as OLE server.

This identification will then be used in the Registry and the Typelibrary to identify the relevant controls. These codes must be inserted manually (guidgen.exe) into the corresponding object and attributes. This UUID is generated via the program guidgen.exe, which has been installed with the Microsoft C++ compiler.

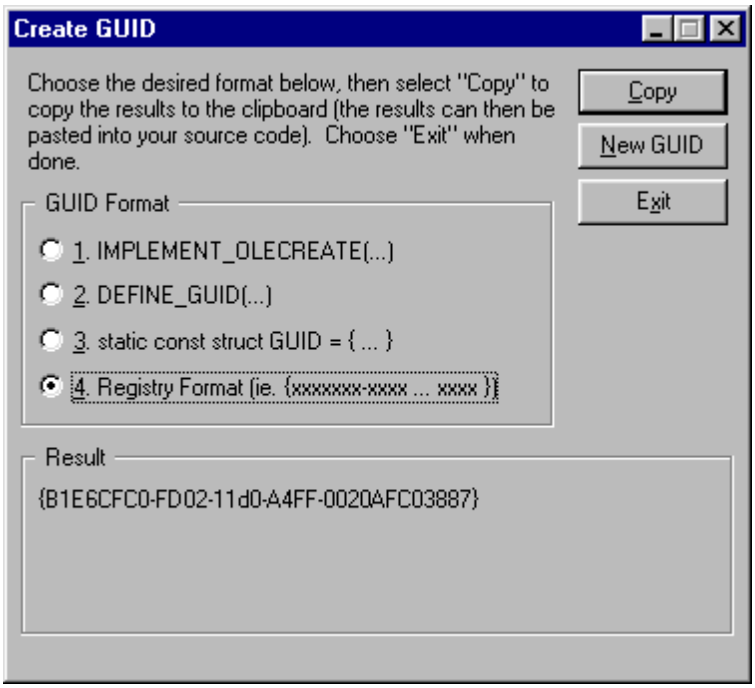

Figure 12: *Generation of GUID*

Example

```
dialog OLEServer
{
    .uuid "A5142E00-F4B7-11d0-AA13-00608C63F57F";
}
message Event;

model control Mcontrol
{
    .mode mode_server;
    .uuid "A5142E01-F4B7-11d0-AA13-00608C63F57F";
    string Property := "";
    rule void Method( string Param)
    {
}
child record PrivateData
```

```
{
}
child groupbox Childobject
{
}
}
```

5.3 Single Usage and Multi-usage of an OLE Server

Depending on how the control object is defined in the Dialog Manager, one OLE server can operate one or more clients at the same time. If the control object is defined as model, several clients can connect to this server. However, if the control object is defined as instance, only one single client can connect to the server. For the next client a new server process will be started.

Both methods have their pros and cons:

- If an individual server process is started for each client, several processes work simultaneously. All these processes have to be maintained in the main memory, something which uses up a lot of main memory resources. The advantage is that there is nothing special to observe when programming. If the dialog is to run as "standalone" dialog without OLE client, the attribute ".mode" of the control object in the dialog-start rule must be modified by assigning the value "mode_none" to the attribute. Then the server is registered in the system and clients can connect to the server.
- If a server is to operate several clients, the entire program must be able to handle changes of the active client at any time, i.e. each time the Dialog Manager enters into the event processing, new OLE calls can be activated for processing. This is why you must not work with global variables in this case (neither in the Rule Language nor in the linked programs). The server must not be finished before having ended all its clients. Usually this is why the active connections to the clients are also counted and the server will be closed, when the number is 0 again.

Example

```
model control MyControl
{
  integer Count := 0;

  on start
  {
      MyControl.Count := MyControl.Count + 1;
  }
  on finish
  {
    MyControl.Count := MyControl.Count - 1;
    if MyControl.Count=0 then
      exit();
    endif
  }
```

```
}
```

In this example the OLE server can be used simultaneously as many times as you want. Each time a new user connects the count is increased; when the user terminates the connection, the count is decreased accordingly. If the count reaches 0, the server will be closed.

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
```

This OLE server can operate only one client at the same time. If a second client wishes to use the services of that server a new server process will be started.

5.4 Attributes

The user-defined attributes of a control object correspond to the so-called properties of the IDispatch name convention. Built-in attributes of the object are not exported; i.e. they cannot be set or get directly from outside. However you can do so by explicitly providing user-defined attributes or methods. Exported attributes must not be generated during runtime of program, but must be defined statically. From these attributes the actual interface information which OLE clients need to call the server is generated.

Example

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
  .width 320;
  .height 200;
  integer I := 123;
  string S := "Dialog Manager";
  boolean B := true;
```

In this example the attributes "I", "S" and "B" can be queried by the client, whereas the attributes ".height" and "width" cannot.

5.5 Methods

User-defined methods of the control object can be accessed by the OLE client; built-in methods such as :insert or :delete are not automatically available to the user using the control object.

Example

```
control PropMethEvent
{
```

```
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  rule void M1
  {
  }
  rule integer M2 (integer I input)
  {
    return 17;
  }
```

In this example the methods "M1" and "M2" can be called by the client.

5.6 Notifications

A PropertyChanged notification is sent to the client when values of user-defined attributes are modified at a control object in the server mode. A client which has not been programmed by Dialog Manager must support notifications via the standard interface IPropertyNotifySink interface.

Example

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
  .width 320;
  .height 200;
  integer I := 123;
  string S := "Dialog Manager";
  boolean B := true;
```

In this example the client receives notifications, when the attributes "I", "S" and "B" are modified, but not when the attributes ".height" and "width" are modified.

5.7 Events

An OLE server can send events to its client. For this purpose the resource ***message*** is available in the Dialog Manager and is defined at the dialog or module. At the control object the developer must specify which messages can be sent. The message object is available also in other object classes and corresponds to the former extevent event.

5.7.1 The message Resource

Events must be defined before using them, including their parameters. For this purpose the resource "message" is used, which is defined as follows:

```
<message> ::= {export} 'message' <label> { <messageSpec> };
<messageSpec> ::= '(' { <messageArg> [ ',' <messageArg> ] } ')'
<messageArg>  ::= <datatype> [<label>]
```

Events to be exported must be defined at the control object, if these are to be passed on to the client. For this purpose the field attribute .message[] is available. This field can be used similarly to .content [], i.e. it can be accessed with setvalue /getvalue and the size can be queried with .count[].

The message resource can also be used independently of OLE control to define "named" events. The event can simply be sent to the control object with a sendevent(). Only those events defined in the field .message[] will be sent to the client, all other events will be processed in the dialog itself, just like external events.

Restrictions

The IDM OLE option supports the data types integer, string and boolean only. Other data types may be defined at the message resource, however it is not possible to generate a type library. Thus these data types cannot be used for data exchange via OLE.

Example

```
dialog D
message Msg(integer I, string S);
model control MC
{
  .message[1] Msg;
  :
  pushbutton Pb
  {
    on select
    {
      sendevent(this.control,Msg,1998,"trigger action");
    }
  }
}
```

Notes and Restrictions

The client has to implement the event interface and link it to NotifySinks via the standard procedure. The client may either generate the corresponding sink dynamically via the type-library of the server, or statically use a sink for a known control. Sending unexpected, new events can produce a crash! This is why you should specify a new control UUID for new events.

5.8 Access to Attributes of any Object

Usually only user-defined attributes of the control can be accessed from an external application. However it is often helpful to access and manipulate certain attributes of the child object or its children. You may program this possibility via methods or via the shadow mechanism of the Dialog

Manager. When you create models, note that the attribute of the instance is "shadowed" and NOT the attribute of the model.

Example

```
dialog D

model control Mcontrol
{
  .mode mode_server;
  string Visible shadows instance GrpBox.visible;
  string Content shadows instance Et.content;
  child groupbox GrpBox
  {
    child edittext Et
    {
    }
  }
}
```

If the control is used from outside in this example, an instance of this model will be generated. When .Visible is accessed the value to be set will be passed on to groupbox GrpBox which will become visible in the container, but only if the container is InPlace-active. With the attribute .Content the .content of textfield Et can be set and read.

Alternatively you may also use methods.

Example

```
dialog D

model control Mcontrol
{
  .mode mode_server;
  rule boolean GetVisible()
  {
    return this.GrpBox.visible;
  }
  rule void SetVisible( boolean Value )
  {
    this.GrpBox.visible := Value;
  }
  rule string GetContent()
  {
    return this.GrpBox.Et.content;
  }
  rule SetContent( string Value )
  {
    this.GrpBox.Et.content := Value;
```

```
  }
  child groupbox GrpBox
  {
    child edittext Et
    {
    }
  }
}
```

This example has the same effect. Here it becomes also clear why shadow instance must be used. In this method the instance of the control including its children are accessed via "this". Dropping "this" has the same effect as dropping the keyword instance at shadows: the objects of the model would be referenced instead of the objects of the current instance.

5.9 Generating Interface Information

This chapter explains the procedures which are necessary to build an OLE server from a dialog with a ***Control*** defined as *ole_server*.

5.9.1 Generating the idl and reg Files

With the **-writeole <base name>** option of the IDM simulation program **idm.exe**, the files required for registration are generated from a dialog script containing ***Controls*** defined as OLE servers. An **idl** and a **reg** file are generated.

The following additional options may be used to influence the generation:

- **+localserver <path of the actual executable>**
- **+helpdir <directory of the help files>**
- **+typelib <path of the type library>**
- **+deficon <path of the default icon>**
- **+proxy <name of the proxy stub>**
- **-userregistry** (since IDM version A.06.01.g)
  Registration for the current user only (under HKEY_CURRENT_USER in the Windows Registry)

With `idm.exe <dialog> -writeole <file>`, the following default entries are written into the registry file **<file>.reg** if they have not been overwritten by one of the options descibed above

```
LocalServer32 = "<path of idm.exe> <dialog> /Automation"
TypeLib = "<file>.tlb"
HelpDir = ""
DefIcon = ""
```

The name of the proxy stub is set to **<base name>.dll** by default.

5.9.2 Server Registration

Double-clicking the **reg** file (provided that **reg** files are linked correctly in the system) or calling **regedit.exe** with the generated **reg** file as an argument registers the control with the system.

Example

```
server.reg server.idl: server.dlg
  $(IDM) server.dlg \
  -localserver "$(IDM) server.dlg -IDMtracefile server.log" \
    -writeole server
  regedit server.reg
```

In this example, the command line for the server is also specified so that it always generates a trace file. Afterward the server is registered with the system.

5.9.3 Further Processing of the idl File

The **idl** file generated using the **-writeole** option must be processed using the MIDL compiler that comes with MICROSOFT VISUAL STUDIO. The MIDL compiler generates a proxy DLL from the **idl** file, which handles the “OLE marshaling” of the interface. The default path is the output name of **-writeole** with the extension “.dll”, but this may be overridden with the option **+proxy**.

Example

```
server.tlb server_p.c server_i.c server.h dlldata.c: server.idl
  midl /ms_ext /app_config /c_ext /tlb server.tlb /Zp1 \
    /env win32 /Os server.idl
```

The files generated with the MIDL compiler must then be compiled with the C compiler and linked to a DLL.

5.10 Example

This example defines a control that provides an “Answer” method and “Visible” and “Content” attributes. These attributes are defined as references to attributes of a child of the control. In this way, the client can also query and change attributes that are not directly defined on the control.

```
dialog ExampleControl
{
  .uuid "FC4D3263-F6DC-11d0-AA13-00608C63F57F";
}

model control MDeepThought
{
  .mode mode_server;
  .uuid "FC4D3264-F6DC-11d0-AA13-00608C63F57F";
```

```
  boolean Visible shadows instance GrpBox.visible;
  string  Content shadows instance EQuestion.content;

  rule integer Answer( string Question )
  {
    this.GrpBox.SAnswer.text := "42";
    return 42;
  }

  child groupbox GrpBox
  {
    child edittext   EQuestion {}
    child statictext SAnswer {}
  }
}
```

This control is registered using the **-writeole** option followed by the **regedit.exe** program. Thereafter, a control is available to the applications, which can display a groupbox with an input field and a text in the control, but only as long as it is activated “InPlace”. Trivially, the answer is always 42.

5.11 Exemplary Integration of a Server in Word 7.0

In Word 7.0 an OLE server which has been built with Dialog Manager is to be integrated with Visual Basic for Applications. The server is only meant to provide methods which may be called by the client.

5.11.1 Server Dialog

The server consists of a listbox which can be displayed within the client area. For this purpose the control object is defined as usual, and the listbox is defined as its child.

```
dialog IDM_Server
{
  .uuid "523E0007-F149-11d0-91F6-00A02444C34E";
}
tile TiCtrl "IDM_IMAGES:client.bmp";
variable integer Refcount := 0;
model control Serv
{
  .mode mode_server;
  .uuid "523E0008-F149-11d0-91F6-00A02444C34E";
  .interfaceid "523E0009-F149-11d0-91F6-00A02444C34E";
  .picture TiCtrl;
  rule void Beep
  {
      this.Lb:Msg("Beep()");
```

```
        beep();
    }
    rule void PPrint (string S input)
    {
        this.Lb:Msg((("Print(" + S) + ")"));
    }
    rule void KKill
    {
        exit();
    }
    on start
    {
        Refcount := (Refcount + 1);
        this.Lb:Msg(("Refcount: " + itoa(Refcount)));
    }
    on finish
    {
        Refcount := (Refcount - 1);
        this.Lb:Msg(("Refcount: " + itoa(Refcount)));
        if (Refcount <= 0) then
        exit();
        endif
    }
    child listbox Lb
    {
      .width 300;
      .height 400;
      .firstchar 1;
      rule void Msg (string S input)
      {
      this.content[(this.itemcount + 1)] := S;
      updatescreen();
      }
    }
}
```

This server is integrated into the system, so that the OLE simulation program "idmole" loads this dialog. Then you can program in Word 97.

5.11.2 Implementing the Client

To implement the client Word 97 is used. Pushbuttons are created to start and close the OLE server. In addition, two more pushbuttons are created to call the methods in the server.

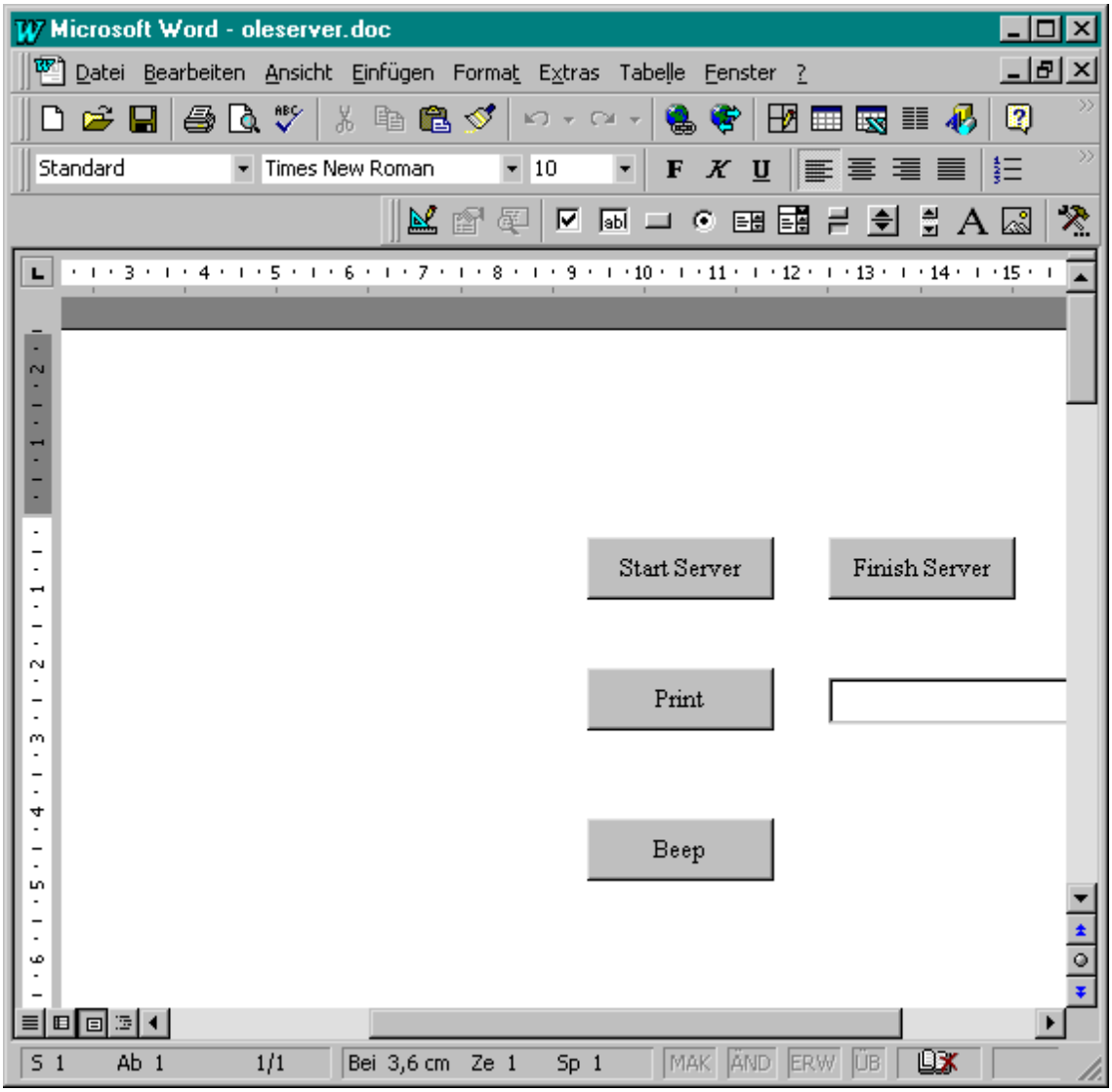

Figure 13: *Client definition in Word 97*

The programming in Visual Basic for applications is as follows:

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "ThisDocument"
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Public OLE As Object

Private Sub EtPrint_Change()

End Sub
```

```
Private Sub PbEnde_Click()
OLE.OLEFormat.Object.KKill
End Sub

Private Sub PbPrint_Click()
OLE.OLEFormat.Object.PPrint EtPrint.Text
End Sub

Private Sub PbBeep_Click()
OLE.OLEFormat.Object.Beep
End Sub

Private Sub PbStart_Click()
Set OLE = ActiveDocument.Shapes.AddOLEObject(ClassType:="IDM_Server.Serv",
Width:=250, Height:=200)

End Sub
```

5.11.3 Working with the Server

After selecting the start pushbutton the server is activated and appears in the client. By selecting the pushbuttons "Beep" and "Print" the corresponding methods can be called in the server.

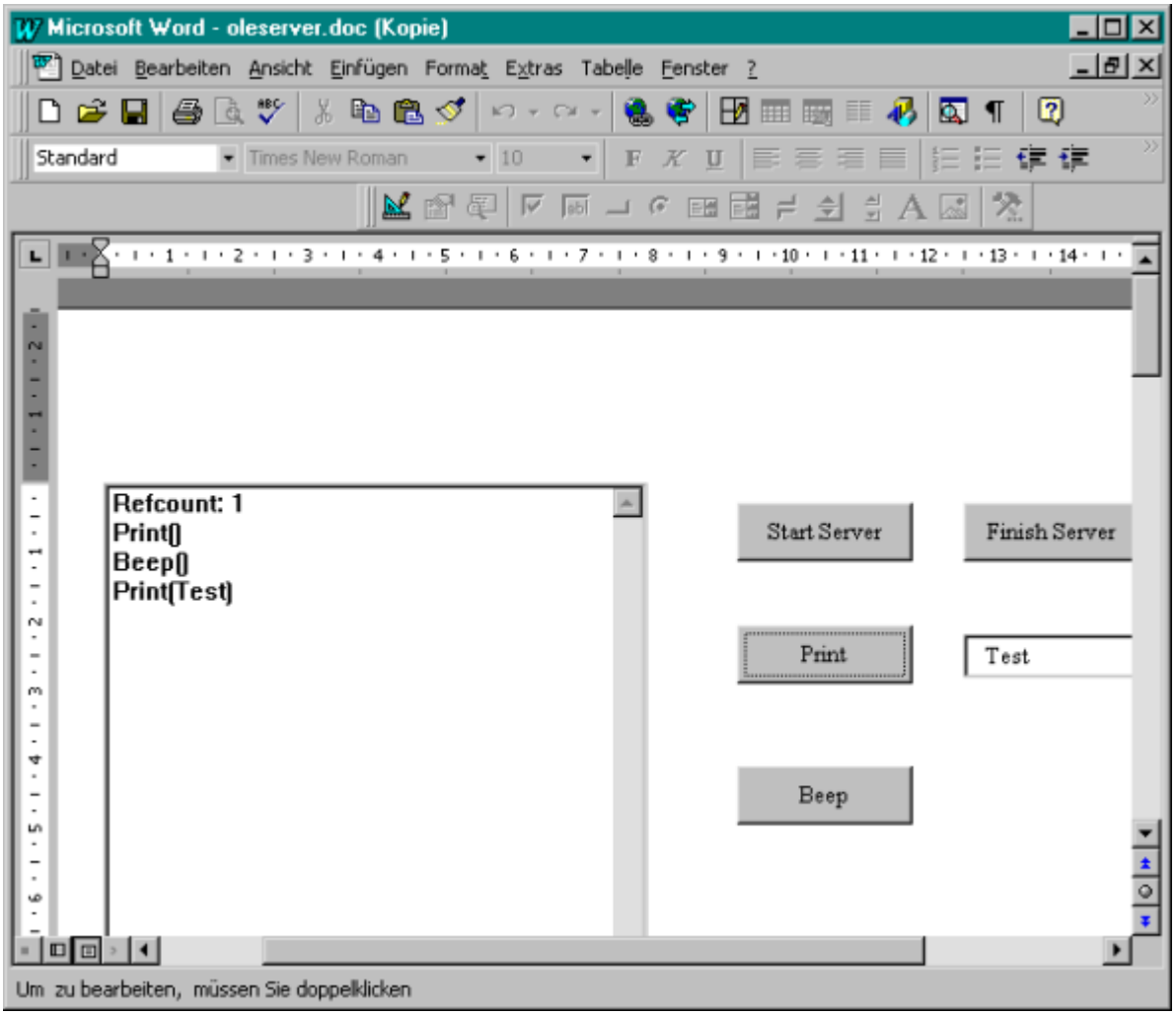

Figure 14: *Word 7.0 client with integrated server*

5.12 Summary of How to Provide an OLE Server

The following steps must be performed when implementing an OLE server:

1. Implement a ***Control*** object.
   - » As model: the server can be used by several clients simultaneously.
   - » As instance: an individual server process is started for each client.
2. Specify a unique UUID with the program **guidgen.exe**.
3. Generate the **idl** and **reg** files with the option **-writeole**.
4. Register the server in the system with the program **regedit.exe**.
5. Generate the necessary C and TBL files with the MIDL compiler.
6. Compile the generated C files.
7. Link a DLL, which is necessary to call the server from the client.

If the OLE server is installed on another computer, the registration must be carried out again and adjusted if necessary, otherwise the OLE server cannot be used.

6 Server and Client Implementation

In this example an OLE client as well as an OLE server will be implemented in the Dialog Manager and the different possibilities for the data exchange are described.

6.1 Client Structure

The client consists of a notebook and an edittext. Via the notebook the server is controlled and in the listbox the results, replies and notifications of the server are traced.

On the first page of the notebook the server is activated and deactivated. On the second page the properties of the server can be queried and set. On the third page the methods of the server can be called.

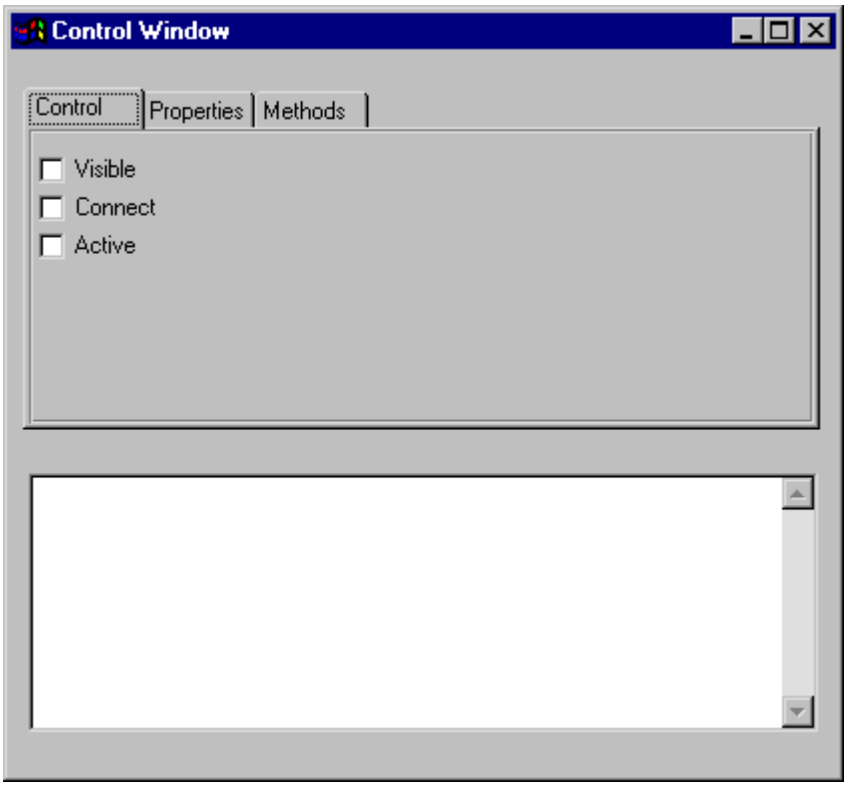

Figure 15: *Client window*

The control object is defined as follows:

```
window WiControl
{
  .title "OLE Control";
  .width 60;
  .height 20;
  .xleft 450;
  .visible := false;
```

```
  control C
  {
    .mode mode_client;
    .name "IdmTest.PropMethEvent";
    .width 300;
    .height 300;
  }
}
```

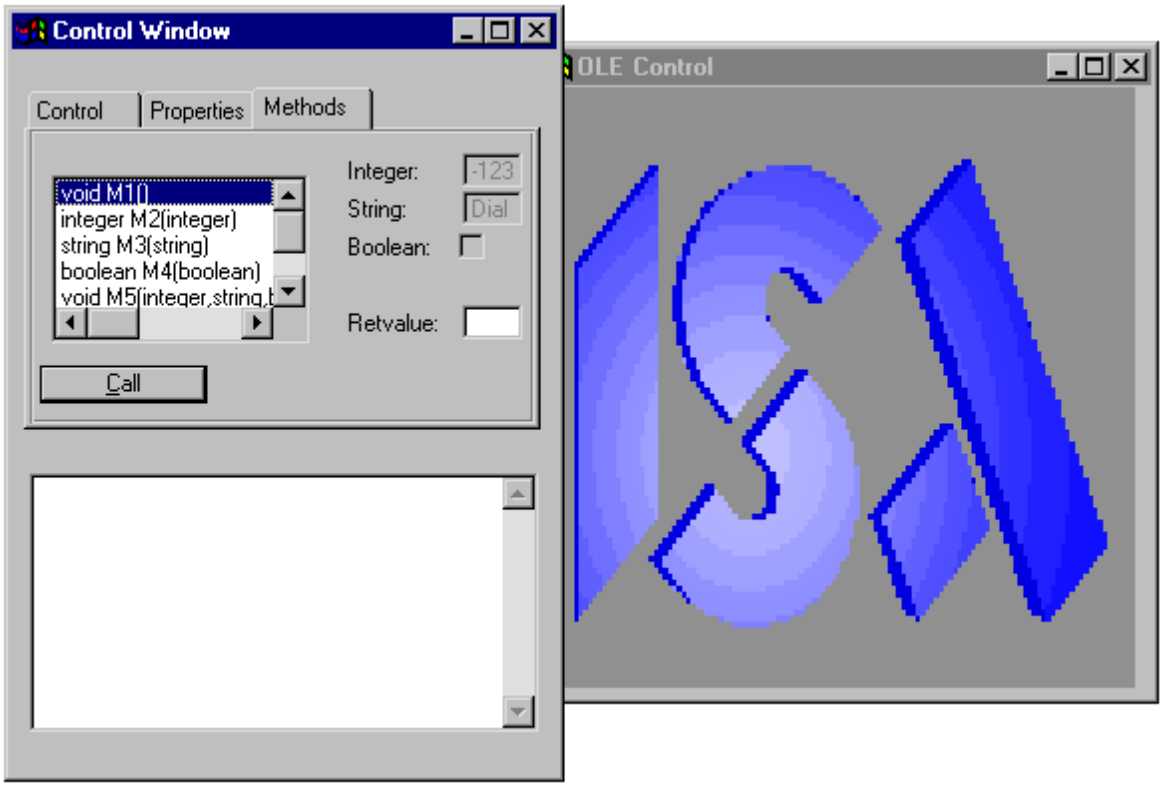

Figure 16: *State of application after starting the server*

6.1.1 Server Activation

By selecting the "visible" checkbox the server is started. By selecting the "connect" checkbox a connection to the server is established and by selecting the "active" checkbox the server is activated and is available for communication.

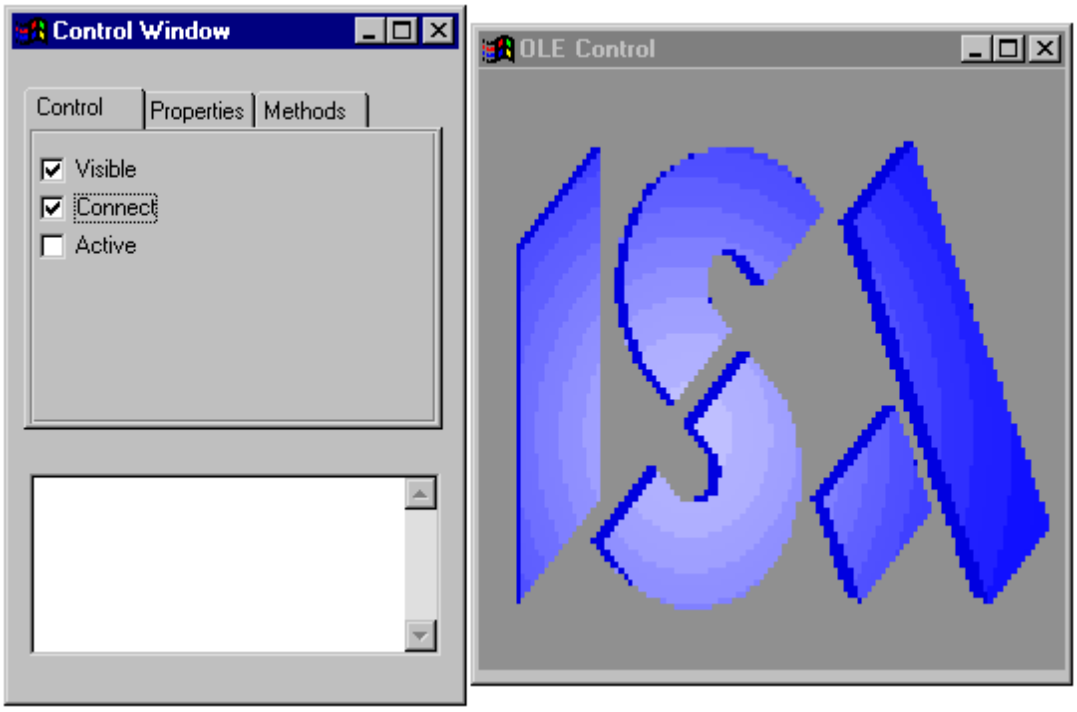

Figure 17: Server after having established the connection

The necessary rules are as follows:

First a general model is defined to define the rules:

```
model checkbox MCb
{
  boolean Value := false;

  on activate,deactivate
  {
    this.Value := this.active;
    if this.Value <> this.active then
      Info("changing "+this.text+" FAILED");
    endif
  }
}
```

The actual definition of the objects is as follows:

```
  child notepage NpControl
  {
     .active true;
     .title "Control";
     MCb CbVisible
     {
       .text "Visible";
       .Value shadows WiControl.visible;
     }
```

```
    MCb CbConnect
    {
      .ytop 1;
      .text "Connect";
      .Value shadows WiControl.C.connect;
    }
    MCb CbActive
    {
      .ytop 2;
      .text "Active";
      .Value shadows WiControl.C.active;
    }
  }
```

6.1.2 Querying and Setting Values in the Server

To query and set values in the server you have to switch to the second page of the notebook, where you have the possibility to query and set attributes.

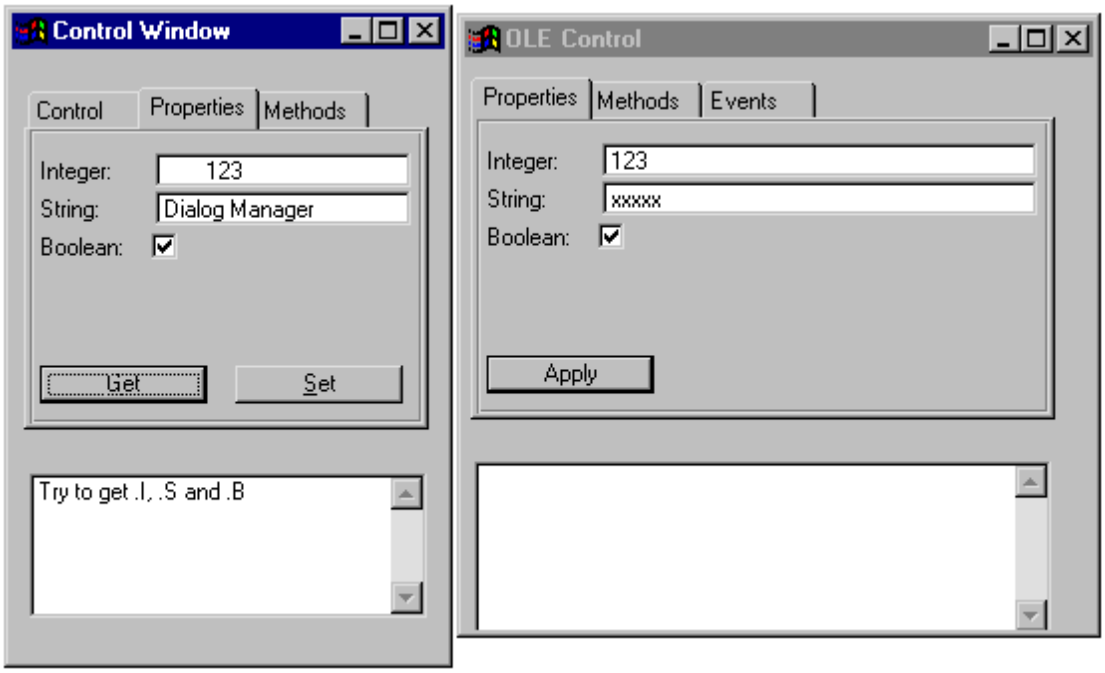

Figure 18: *Querying and setting values in the server*

The rules that are necessary in the client are as follows:

```
child pushbutton PbGet
{
  .yauto -1;
  .height 1;
  .text "&Get";
  .defbutton true;
```

```
  on select
  {
    Info("Try to get .I, .S and .B");
    if fail(this.parent.Integer.Value := this.window.C.I) then
      Info("getvalue of .I FAILED");
    endif
    if fail(this.parent.String.Value := this.window.C.S) then
      Info("getvalue of .S FAILED");
    endif
    if fail(this.parent.Boolean.Value := this.window.C.B) then
      Info("getvalue of .B FAILED");
    endif
  }
}
child pushbutton PbSet
{
  .yauto -1;
  .height 1;
  .text "&Set";
  .xleft 14;
  on select
  {
    Info("Try to set .I, .S and .B");
    if fail(this.window.C.I := this.parent.Integer.Value) then
      Info("setvalue of .I FAILED");
    endif
    if fail(this.window.C.S := this.parent.String.Value) then
      Info("setvalue of .S FAILED");
    endif
    if fail(this.window.C.B := this.parent.Boolean.Value) then
      Info("setvalue of .B FAILED");
    endif
  }
}
```

As you can see here, the attributes "I", "S" and "B" of the control object are accessed in the rules, although these are not defined there. These attributes are passed on to the server and are processed there.

6.1.3 Reaction to Events

If the server can send notifications to the client, these notifications must be caught up at the client. The client can thus react to these events which will arrive in the Rule Language as "external events" and which must be programmed accordingly. If these events are to receive parameters from the server, the events must be defined as parameters of the rule.

```
control C
```

```
{
    on extevent "Msg1"
    {
      Info("extevent Msg1()");
    }
    on extevent "Msg2" (integer I)
    {
      Info("extevent Msg2("+I+")");
    }
    on extevent "Msg3" (string S)
    {
      Info("extevent Msg3("+S+")");
    }
    on extevent "Msg4" (boolean B)
    {
      Info("extevent Msg4("+B+")");
    }
    on extevent "Msg5" (integer I, string S, boolean B)
    {
      Info("extevent Msg5("+I+","+S+","+B+")");
    }
    on extevent "Msg6" (integer P1, string P2, boolean P3,
    integer P4, string P5, boolean P6)
    {
      Info("extevent Msg6("+P1+","+P2+","+P3+","+P4+","
         +P5+","+P6+")");
    }
}
```

6.1.4 Reaction to Notifications

If the server is able to send notifications to the client, they can be caught up at the client. Then the client can react to the modifications of values. In the Rule Language these notifications arrive as "changed events" and must be programmed accordingly.

```
control C
{
    on .I changed
    {
      Info(".I changed");
if fail(NpProperties.Integer.Value := this.I) then
        Info("getvalue of .I FAILED");
      endif
    }
    on .S changed
    {
      Info(".S changed");
```

```
  if fail(NpProperties.String.Value := this.S) then
        Info("getvalue of .S FAILED");
      endif
    }
    on .B changed
    {
      Info(".B changed");
  if fail(NpProperties.Boolean.Value := this.B) then
        Info("getvalue of .B FAILED");
      endif
    }
}
```

6.1.5 Calling Methods in the Server

The call of methods including its corresponding parameters must be implemented in the client, so that methods of the server may be called from the client. The method itself must not be defined.

The rule code for calling methods at the server is as follows:

```
child pushbutton PbCall
  {
    .yauto -1;
    .height 1;
    .text "&Call";
    .defbutton true;
    integer I shadows instance NpMethods.Integer.Value;
    boolean B shadows instance NpMethods.Boolean.Value;
    string S shadows instance NpMethods.String.Value;
    on select
    {
      case this.parent.LbMethods.activeitem
    in 1:
      Info("call :M1();");
      this.parent.Retval.Value := "";
      if fail(this.window.C:M1()) then
      Info("FAILED");
      endif
    in 2:
      Info("call :M2("+this.I+");");
      this.parent.Retval.Value := "";
      if fail(this.parent.Retval.Value:=
          ""+this.window.C:M2(this.I)) then
         Info("FAILED");
      endif

    in 3:
```

```
    Info("call :M3("+this.S+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
       ""+this.window.C:M3(this.S)) then
       Info("FAILED");
    endif

  in 4:
    Info("call :M4("+this.B+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
      ""+this.window.C:M4(this.B)) then
      Info("FAILED");
    endif

  in 5:
    Info("call :M5("+this.I+","+this.S+","+this.B+");");
    this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
        ""+this.window.C:M5(this.I,this.S,this.B)) then
        Info("FAILED");
    endif
  in 6:
    Info("call :M6("+this.I+","+this.S+","+this.B+","+
     this.I+","+this.S+","+this.B+","+this.I+","+
      this.S+");");
       this.parent.Retval.Value := "";
    if fail(this.parent.Retval.Value:=
      ""+this.window.C:M6(this.I,this.S,this.B,this.I,this.S,
      this.B,this.I,this.S)) then
      Info("FAILED");
    endif

  otherwise:
    Info("Error - unknown method");
      endcase
  }
}
```

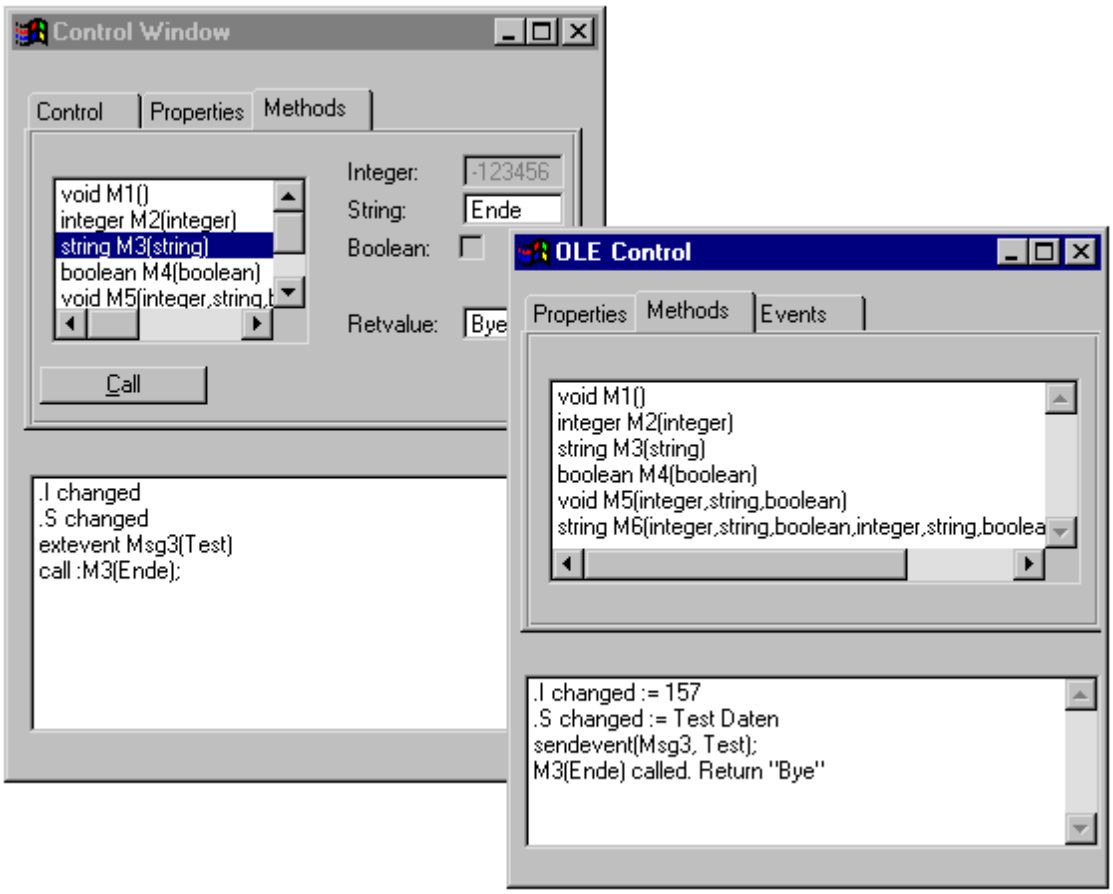

Figure 19: *Calling methods in the OLE server*

6.1.6 The Client Dialog

```
dialog Client
{
}
color ColWin
{
    0: rgb(47,175,207), grey(255), white;
    1: rgb(192,192,192), grey(255), white;
}
color ColInput
{
    0: rgb(111,159,175), grey(200), white;
    1: rgb(255,255,192), grey(200), white;
}
color ColBlack "BLACK", grey(0), white;
color ColWhite "WHITE", grey(0), white;
color ColRed "RED", grey(0), white;
color ColGreen "GREEN", grey(0), white;
color ColBlue "BLUE", grey(0), white;
color ColYellow "YELLOW", grey(0), white;
```

```
font FontNormal "8.Helv";
font FontBig "14.Helv";
font FontFixed "12.System VIO";

model groupbox MInteger
{
  .height 1;
  integer Value := -123456789;
  .xauto 0;
  on .Value changed
  {
    this.Et.content := itoa(this.Value);
  }
  child statictext
  {
    .text "Integer:";
  }
  child edittext Et
  {
    .xleft 8;
    .format "%-9d";
    .xauto 0;
    .content "-123456789";
    on charinput
    {
    if fail(this.parent.Value := atoi(this.content)) then
        this.parent.Value := 0;
    endif
    }
  }
}
model groupbox MString
{
  .xauto 0;
  .height 1;
  string Value shadows instance MString.Et.content;
  string Text shadows instance MString.St.text;
  child statictext St
  {
    .text "String:";
  }
  child edittext Et
  {
    .xleft 8;
    .xauto 0;
    .content "Dialog Manager";
```

```
  }
}
model groupbox MBoolean
{
  .height 1;
  boolean Value shadows instance Cb.active;
  .width 80;
  .height 20;
  child statictext
  {
    .text "Boolean:";
  }
  child checkbox Cb
  {
    .xleft 8;
    .text "";
  }
}

window WiControl
{
  .title "OLE Control";
  .width 60;
  .height 20;
  .xleft 450;
  .visible := false;

  control C
  {
    .mode mode_client;
    .name "IdmTest.PropMethEvent";
     .width 300;
    .height 300;
    on .visible changed
    {
     CbVisible.Value := this.visible;
    }
    on .connect changed
    {
     CbConnect.Value := this.connect;
    }
    on .active changed
    {
     CbActive.Value := this.active;
    }
    on extevent "Msg1"
```

```
{
 Info("extevent Msg1()");
}
on extevent "Msg2" (integer I)
{
  Info("extevent Msg2("+I+")");
}
on extevent "Msg3" (string S)
{
  Info("extevent Msg3("+S+")");
}
on extevent "Msg4" (boolean B)
{
  Info("extevent Msg4("+B+")");
}
on extevent "Msg5" (integer I, string S, boolean B)
{
 Info("extevent Msg5("+I+","+S+","+B+")");
}
on extevent "Msg6" (integer P1, string P2, boolean P3,
   integer P4, string P5, boolean P6)
{
 Info("extevent Msg6 ("+P1+","+P2+","+P3+","+P4+"," +P5+","
    +P6+")");
}
on .I changed
{
  Info(".I changed");
  if fail(NpProperties.Integer.Value := this.I) then
    Info("getvalue of .I FAILED");
  endif
}
on .S changed
{
  Info(".S changed");
  if fail(NpProperties.String.Value := this.S) then
    Info("getvalue of .S FAILED");
  endif
}
on .B changed
{
  Info(".B changed");
  if fail(NpProperties.Boolean.Value := this.B) then
    Info("getvalue of .B FAILED");
  endif
}
```

```
  }
}
rule void Info(string S)
{
  LbInfo.content[LbInfo.itemcount+1]  := S;
  LbInfo.topitem := LbInfo.itemcount;
}
model checkbox MCb
{
  boolean Value := false;

  on activate,deactivate
  {
    this.Value := this.active;
    if this.Value <> this.active then
      Info("changing "+this.text+" FAILED");
    endif
  }
}
window WiMain
{
  .width 60;
  .height 20;
  .title "Control Window";

  on close
  {
    exit();
  }
  object C shadows WiControl.C.self;

  child groupbox Gb
  {
    .xauto 0;
    .yauto 0;
    .borderwidth 0;
    child notebook Nb
    {
      .xauto 0;
      .yauto 1;
      .height 10;
    child notepage NpControl
    {
      .active true;
       .title "Control";
       MCb CbVisible
```

```
 {
   .text "Visible";
   .Value shadows WiControl.visible;
 }
   MCb CbConnect
 {
   .ytop 1;
   .text "Connect";
   .Value shadows WiControl.C.connect;
 }
   MCb CbActive
 {
   .ytop 2;
   .text "Active";
   .Value shadows WiControl.C.active;
 }
}

  child notepage NpProperties
  {
    .title "Properties";
    child MInteger Integer
    {
    }
    child MString String
    {
      .ytop 1;
    }
    child MBoolean Boolean
    {
      .ytop 2;
    }
    child pushbutton PbGet
    {
      .yauto -1;
      .height 1;
      .text "&Get";
      .defbutton true;
      on select
      {
        Info("Try to get .I, .S and .B");
        if fail(this.parent.Integer.Value :=
           this.window.C.I) then
             Info("getvalue of .I FAILED");
        endif
        if fail(this.parent.String.Value :=
```

```
          this.window.C.S) then
            Info("getvalue of .S FAILED");
          endif
          if fail(this.parent.Boolean.Value :=
            this.window.C.B) then
            Info("getvalue of .B FAILED");
          endif
        }
      }
      child pushbutton PbSet
      {
        .yauto -1;
        .height 1;
        .text "&Set";
        .xleft 14;
        on select
          {
          Info("Try to set .I, .S and .B");
            if fail(this.window.C.I :=
               this.parent.Integer.Value) then
                Info("setvalue of .I FAILED");
            endif
            if fail(this.window.C.S :=
               this.parent.String.Value) then
               Info("setvalue of .S FAILED");
            endif
            if fail(this.window.C.B :=
               this.parent.Boolean.Value) then
              Info("setvalue of .B FAILED");
            endif
       }
     }
  }
    child notepage NpMethods
    {
       .title "Methods";
        child listbox LbMethods
      {
        .xauto 1;
        .width 20;
        .yauto 0;
        .ybottom 1;
        .content[1] "void M1()";
        .content[2] "integer M2(integer)";
        .content[3] "string M3(string)";
        .content[4] "boolean M4(boolean)";
```

```
        .content[5] "void M5(integer,string,boolean)";
        .content[6] "string    M6(integer,string,boolean,
                   integer,string,boolean,integer,string)";
        .activeitem 1;
        .firstchar 1;
        on select
        {
       this.parent.Integer.sensitive :=
  (0 <> stringpos(this.content[this.activeitem], "integer"));
      this.parent.String.sensitive :=
  (0 <> stringpos(this.content[this.activeitem], "string"));
      this.parent.Boolean.sensitive :=
  (0 <> stringpos(this.content[this.activeitem], "boolean"));
        }
   }
child MInteger Integer
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .Et.active false;
}
child MString String
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 1;
}
child MBoolean Boolean
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 2;
}
child MString Retval
{
  .Text := "Retvalue:";
  .Value := "";
  .xleft 22;
  .yauto 1;
  .ytop 4;
}
child pushbutton PbCall
{
```

```
.yauto -1;
.height 1;
.text "&Call";
.defbutton true;
integer I shadows instance NpMethods.Integer.Value;
boolean B shadows instance NpMethods.Boolean.Value;
string S shadows instance NpMethods.String.Value;
on select
{
  case this.parent.LbMethods.activeitem
in 1:
  Info("call :M1();");
  this.parent.Retval.Value := "";
  if fail(this.window.C:M1()) then
  Info("FAILED");
  endif
in 2:
  Info("call :M2("+this.I+");");
  this.parent.Retval.Value := "";
  if fail(this.parent.Retval.Value:=
      ""+this.window.C:M2(this.I)) then
     Info("FAILED");
  endif

in 3:
  Info("call :M3("+this.S+");");
  this.parent.Retval.Value := "";
  if fail(this.parent.Retval.Value:=
     ""+this.window.C:M3(this.S)) then
     Info("FAILED");
  endif

in 4:
  Info("call :M4("+this.B+");");
  this.parent.Retval.Value := "";
  if fail(this.parent.Retval.Value:=
    ""+this.window.C:M4(this.B)) then
    Info("FAILED");
  endif

in 5:
  Info("call :M5("+this.I+","+this.S+","+this.B+");");
  this.parent.Retval.Value := "";
  if fail(this.parent.Retval.Value:=
      ""+this.window.C:M5(this.I,this.S,this.B)) then
      Info("FAILED");
```

```
      endif
    in 6:
      Info("call :M6("+this.I+","+this.S+","+this.B+","+
       this.I+","+this.S+","+this.B+","+this.I+","+
        this.S+");");
         this.parent.Retval.Value := "";
      if fail(this.parent.Retval.Value:=
        ""+this.window.C:M6(this.I,this.S,this.B,this.I,this.S,
        this.B,this.I,this.S)) then
        Info("FAILED");
      endif

    otherwise:
      Info("Error - unknown method");
        endcase
    }
  }
  }
  }
  }
  child listbox LbInfo
  {
    .xauto 0;
    .yauto 0;
    .ytop 10;
    .firstchar 1;
  }
}
```

6.2 Server Structure

The server also consists of a ***notebook*** and a ***listbox***. The server is controlled via the ***notebook*** and in the ***listbox*** the results and requests of the client are logged. The server is operated “InPlace”, that is, it uses the area within a window of the client for its presentation.

If there is a connection to the server but the server has not yet been activated, a picture is displayed in the client area. After activation, the objects of the server appear in the client.

In order for the server to be executable, the ***dialog*** and the ***control*** object have each been assigned a unique UUID.

```
dialog IdmTest
{
  .uuid "499593d0-a159-11d1-a7e3-00a02444c34e";
}
```

```
tile TiPropMethEvent "IMD_IMAGES:isaicon.gif";

default control CONTROL
{
  integer Count := 0;

  on start
  {
    CONTROL.Count := CONTROL.Count + 1;
  }

  on finish
  {
    CONTROL.Count := CONTROL.Count - 1;
    if CONTROL.Count=0 then
      exit();
    endif
  }
}

control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
}
```

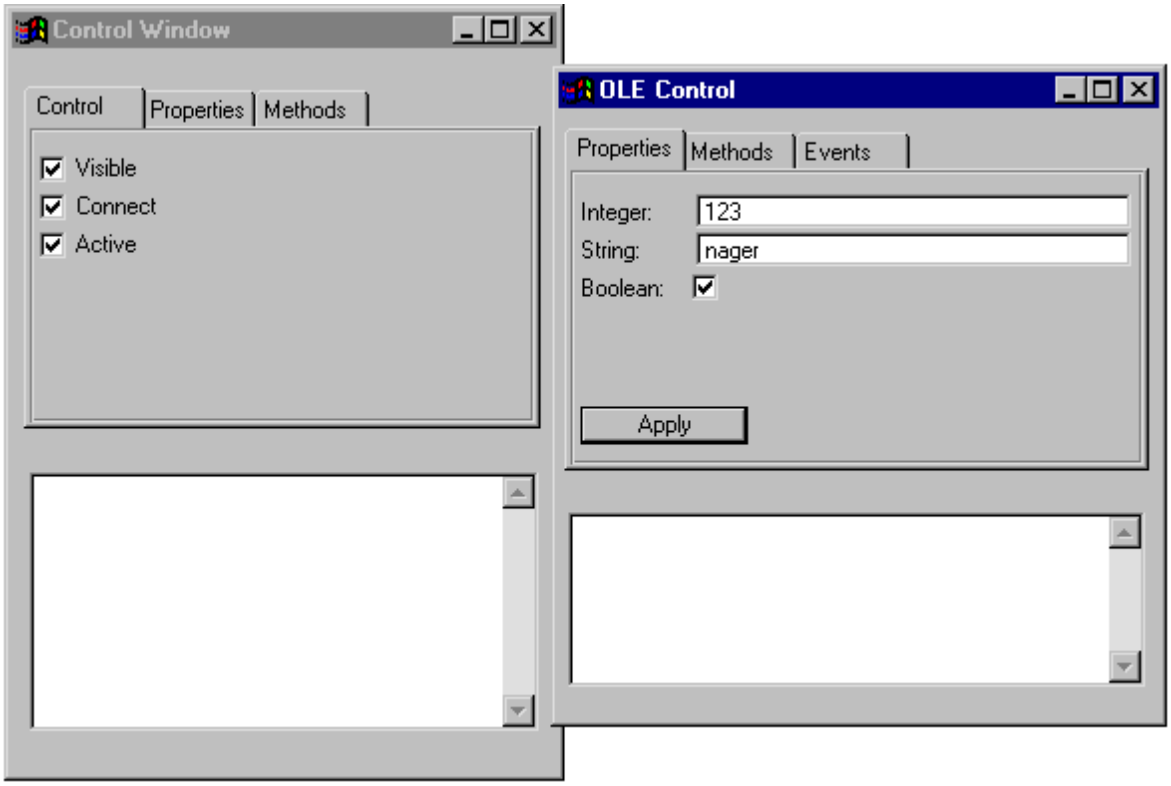

Figure 20: *Active OLE server*

6.2.1 Providing the Server

Unlike the client, some steps must be taken at the server to allow the server to operate as an OLE server.

Initially, the **-writeole** option of the IDM simulation program is used to generate the **reg** file necessary to register the OLE server and the **idl** file necessary to provide the runtime components. The command line looks like this:

```
$(THIS).reg $(THIS).idl: $(THIS).dlg
  $(IDM) $(THIS).dlg -localserver "$(IDM) $(THIS).dlg \
    -IDMenv MODLIB=$(THISDIR) -IDMerrfile $(THIS).log" \
    -writeole $(THIS)
  regedit $(THIS).reg
```

The `regedit` command directly registers the server in the system.

Then, the generated **idl** file is compiled using the MIDL compiler.

```
$(THIS).tlb $(THIS)_p.c $(THIS).h dlldata.c: $(THIS).idl
  midl /ms_ext /app_config /c_ext /tlb $(THIS).tlb /Zp1 \
    /env win32 /Os $(THIS).idl
```

Finally, a DLL is generated from the object files.

```
$(OUTFILE) : $(OBJS) $(TARGET).res $(DEFFILE)
  echo ++++++++++
  echo Linking $@
  echo $(LINK)                  > $(TARGET).lrf
  echo $(ENTRY)                >> $(TARGET).lrf
  echo -def:$(THIS).def        >> $(TARGET).lrf
  echo -out:$(OUTFILE)         >> $(TARGET).lrf
  echo -machine:IX86           >> $(TARGET).lrf
  echo -subsystem:windows5.01  >> $(TARGET).lrf
  echo -align:0x1000           >> $(TARGET).lrf
  echo $(OBJS1)                >> $(TARGET).lrf
  echo $(OBJS2)                >> $(TARGET).lrf
  echo $(OBJS3)                >> $(TARGET).lrf
  echo $(OBJS4)                >> $(TARGET).lrf
  echo $(OBJS5)                >> $(TARGET).lrf
  echo $(OBJS6)                >> $(TARGET).lrf
  echo $(TARGET).res           >> $(TARGET).lrf
  echo $(LIBS)                 >> $(TARGET).lrf
  echo $(LIBS32)               >> $(TARGET).lrf
  link @$(TARGET).lrf
  del $(TARGET).lrf
```

After that, the server can be used by a client.

The corresponding makefile looks like this:

```
THISDIR=h:\ole\clntserv
THIS=$(THISDIR)\server
IDM=idmole.exe

CL32    = -G3s
WX      =
LINKDLL = /DLL
DEFDLL  = -D_DLL
ENTRY   = -entry:LibMain32

DEFUNICODE = -DWIN32ANSI
LINKD32    = -debug:full $(LINKDLL) -debugtype:cv
LINKN32    = -debug:none $(LINKDLL)
DEFS32     = -DWIN32 $(DEFDLL) -D_X86_=1 $(DEFUNICODE)
TLBDEFS    = -DWIN32
#BOOKLIB   = inole.lib
LIBS32A    = msvcrt.lib kernel32.lib user32.lib gdi32.lib comdlg32.lib
advapi32.lib
LIBS32B    = ole32.lib oleaut32.lib uuid.lib
LIBS32     = $(LIBS32A) $(LIBS32B) $(BOOKLIB)

LIBS   = rpcrt4.lib
```

```
CONTIN =

CFLAGS = -c -Od -Z7 -Ze -W3 -nologo $(CL32) \
         -D_WIN32_WINNT=0x400
LINK   = $(LINKD32) /NOD
DEFS   = $(DEFS32) -DSTRICT -DDEBUG

.SUFFIXES: .h .obj .exe .dll .cpp .res .rc .tlb .odl

OUTFILE = $(THIS).dll
TARGET  = $(THIS)
DEFFILE = $(THIS).def

OBJS1 = $(THIS)_i.obj $(THIS)_p.obj
OBJS2 = dlldata.obj libmain.obj
OBJS3 = ""
OBJS4 = ""
OBJS5 = ""
OBJS6 = ""
OBJS  = $(OBJS1) $(OBJS2)

#####################################################################

.c.obj:
  echo ++++++++++
  echo Compiling $*.c
  cl $(CFLAGS) $(DEFS)  $*.c

.cpp.obj:
  echo ++++++++++
  echo Compiling $*.c
  cl $(CFLAGS) $(DEFS)  $*.cpp

.rc.res:
  echo ++++++++++
  echo Compiling Resources
  rc -r $(DEFS) $(DOC) -fo$@ $*.rc

#####################################################################

all: $(THIS).reg $(THIS).idl $(THIS).tlb $(THIS).dll

clean:
  del $(THIS).tlb
  del $(THIS).reg
  del $(THIS).dll
  del $(THIS).odl
```

```
  del $(THIS)_i.c
  del $(THIS)_p.c
  del $(THIS).h
  del dlldata.c
  del *.obj
  del *.exe
  del *.lrf

$(THIS).reg $(THIS).idl: $(THIS).dlg
  $(IDM) $(THIS).dlg -localserver "$(IDM) $(THIS).dlg \
    -IDMenv MODLIB=$(THISDIR) -IDMerrfile $(THIS).log" \
    -writeole $(THIS)
  regedit $(THIS).reg

$(THIS).tlb $(THIS)_p.c $(THIS).h dlldata.c: $(THIS).idl
  midl /ms_ext /app_config /c_ext /tlb $(THIS).tlb /Zp1 \
    /env win32 /Os $(THIS).idl

$(THIS)_p.obj: $(THIS)_p.c
$(THIS)_i.obj: $(THIS)_i.c
dlldata.obj: dlldata.c
libmain.obj: libmain.cpp

#####################################################################

$(OUTFILE) : $(OBJS) $(TARGET).res $(DEFFILE)
  echo ++++++++++
  echo Linking $@
  echo $(LINK)                  > $(TARGET).lrf
  echo $(ENTRY)                >> $(TARGET).lrf
  echo -def:$(THIS).def        >> $(TARGET).lrf
  echo -out:$(OUTFILE)         >> $(TARGET).lrf
  echo -machine:IX86           >> $(TARGET).lrf
  echo -subsystem:windows5.01  >> $(TARGET).lrf
  echo -align:0x1000           >> $(TARGET).lrf
  echo $(OBJS1)                >> $(TARGET).lrf
  echo $(OBJS2)                >> $(TARGET).lrf
  echo $(OBJS3)                >> $(TARGET).lrf
  echo $(OBJS4)                >> $(TARGET).lrf
  echo $(OBJS5)                >> $(TARGET).lrf
  echo $(OBJS6)                >> $(TARGET).lrf
  echo $(TARGET).res           >> $(TARGET).lrf
  echo $(LIBS)                 >> $(TARGET).lrf
  echo $(LIBS32)               >> $(TARGET).lrf
  link @$(TARGET).lrf
  del $(TARGET).lrf
```

6.2.2 Querying and Setting Attributes

In order that the client can query attributes on the server, nothing needs to be programmed in the server. All attributes defined on the ***control*** object can be requested and set automatically by the client.

In the example, the attributes “I”, “S” and “B” can be queried by the client.

```
control PropMethEvent
{
  integer I := 123;
  string  S := "Dialog Manager";
  boolean B := true;
}
```

6.2.3 Calling Methods

For the client to call methods in the server, these must be defined as normal methods of the ***control*** object. The methods can then access all objects and attributes defined in the dialog.

In this example, the methods look like this:

```
control PropMethEvent
{
  .message[1] Msg1;
  .message[2] Msg2;
  .message[3] Msg3;
  .message[4] Msg4;
  .message[5] Msg5;
  .message[6] Msg6;

  rule void M1
  {
    Info("M1() called.");
    sendevent(this, Msg1);
  }

  rule integer M2 (integer I input)
  {
    Info("M2(" + I + ") called. Return -123456789");
    return -123456789;
  }

  rule string M3 (string S input)
  {
    Info("M3(" + S + ") called. Return \"Bye\"");
    return "Bye";
  }
```

```
  rule boolean M4 (boolean B input)
  {
    Info("M4(" + B + ") called. Return " + ( not B ));
    return ( not B );
  }

  rule void M5 (integer I input, string S input, boolean B input)
  {
    Info("M5(" + I + ", " + S + ", " + B + ") called.");
  }

  rule string M6 (integer P1 input, string P2 input,
    boolean P3 input, integer P4 input, string P5 input,
    boolean P6 input, integer P7 input, string P8 input)
  {
    Info("M6(" + P1 + ", " + P2 + ", " + P3 + ", " + P4 +
         ", " + P5 + ", " + P6 + ", " + P7 + ", " + P8 +
         ") called. Return \"Abracadabra!\"");
    return "Abracadabra!";
  }
}
```

6.2.4 Sending Events

If the server shall send events to the client, these must be defined in the server and programmed accordingly. The definition of such events is done using the ***message*** resource. The events are then sent to the client with the built-in function **sendevent**.

These events are defined in the example as follows:

```
message Msg1;
message Msg2 (integer I);
message Msg3 (string S);
message Msg4 (boolean B);
message Msg5 (integer I, string S, boolean B);
message Msg6 (integer P1, string P2, boolean P3,
              integer P4, string P5, boolean P6);
```

In addition, the events to be sent by a control must be defined on the ***control*** object in the attribute *.message[integer]*. The ***messages*** defined in this attribute are then passed on to the client.

```
control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
```

```
  .message[1] Msg1;
  .message[2] Msg2;
  .message[3] Msg3;
  .message[4] Msg4;
  .message[5] Msg5;
  .message[6] Msg6;
}
```

The sending of the events takes place on selection of the pushbutton “Send” in the following rule:

```
child pushbutton PbSend
{
  .yauto     -1;
  .height    1;
  .text      "Send";
  .defbutton true;
  integer I shadows instance NpEvents.Integer.Value;
  boolean B shadows instance NpEvents.Boolean.Value;
  string  S shadows instance NpEvents.String.Value;

  on select
  {
    case this.parent.LbEvents.activeitem
    in 1:
      sendevent(this.control, Msg1);
      Info("sendevent(Msg1);");
    in 2:
      sendevent(this.control, Msg2, this.I);
      Info("sendevent(Msg2, " + this.I + ");");
    in 3:
      sendevent(this.control, Msg3, this.S);
      Info("sendevent(Msg3, " + this.S + ");");
    in 4:
      sendevent(this.control, Msg4, this.B);
      Info("sendevent(Msg4, " + this.B + ");");
    in 5:
      sendevent(this.control, Msg5, this.I, this.S, this.B);
      Info("sendevent(Msg5, " + this.I + ", " + this.S + ", " +
           this.B + ");");
    in 6:
      sendevent(this.control, Msg6, this.I, this.S, this.B,
                this.I, this.S, this.B);
      Info("sendevent(Msg6, " + this.I + ", " + this.S + ", " +
           this.B + ", " + this.I + ", " + this.S + ", " +
           this.B + ");");
    otherwise:
      Info("Error - unknown event");
    endcase
```

```
  }
}
```

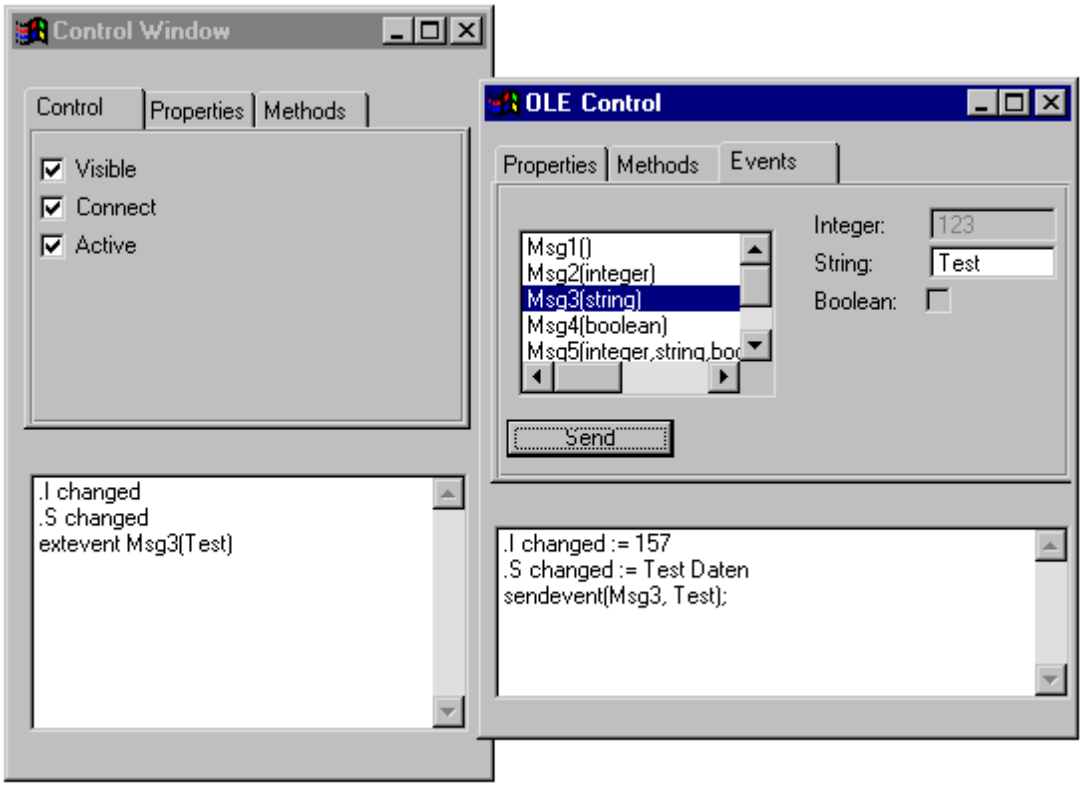

Figure 21: Sending events

6.2.5 Sending Notifications

In order to send notifications about attribute changes to the client, nothing has to be programmed in the server. As soon as an attribute changes on the control, a notification is automatically sent to the client.

The rule code for the assignment within the server looks like this:

```
child pushbutton PbApply
{
  .yauto     -1;
  .height    1;
  .text      "Apply";
  .defbutton true;

  on select
  {
    this.control.I := this.parent.Integer.Value;
    this.control.S := this.parent.String.Value;
    this.control.B := this.parent.Boolean.Value;
```

```
  }
}
```

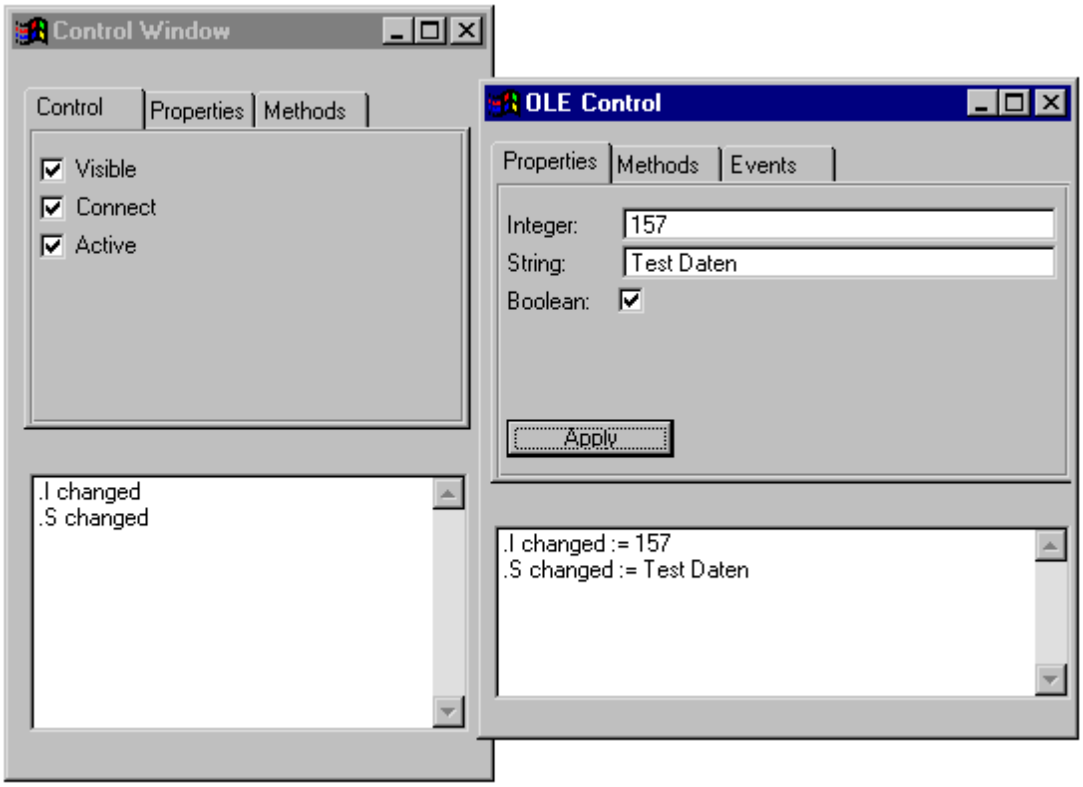

Figure 22: Exchange of notifications on selection of the Apply pushbutton

6.2.6 The Server Dialog

```
dialog IdmTest
{
  .uuid "499593d0-a159-11d1-a7e3-00a02444c34e";
}

tile TiPropMethEvent "IMD_IMAGES:isaicon.gif";

message Msg1;
message Msg2(integer I);
message Msg3(string S);
message Msg4(boolean B);
message Msg5(integer I, string S, boolean B);
message Msg6(integer P1, string P2, boolean P3,
             integer P4, string P5, boolean P6);

model groupbox MInteger
{
```

```
  .height 1;
  .xauto 0;
  integer Value := 123;

  child statictext { .text "Integer:"; }

  child edittext Et
  {
    .xleft 8;
    .xauto 0;
    .format "%-9d";
    .content "123";

    on charinput
    {
      if fail(this.parent.Value := atoi(this.content)) then
        this.parent.Value := 0;
      endif
    }
  }
  on .Value changed
  {
    this.Et.content := itoa(this.Value);
  }
}

model groupbox MString
{
  .height 1;
  .xauto 0;
  string Value shadows instance MString.Et.content;

  child statictext { .text "String:"; }

  child edittext Et
  {
    .xleft 8;
    .xauto 0;
    .content "Dialog Manager";
  }
}

model groupbox MBoolean
{
  .height 1;
  boolean Value shadows instance Cb.active;
```

```
  child statictext { .text "Boolean:"; }

  child checkbox Cb
  {
    .xleft 8;
    .text "";
  }
}

default control CONTROL
{
  integer Count := 0;

  on start
  {
    CONTROL.Count := CONTROL.Count + 1;
  }

  on finish
  {
    CONTROL.Count := CONTROL.Count - 1;
    if CONTROL.Count=0 then
      exit();
    endif
  }
}

model control PropMethEvent
{
  .mode mode_server;
  .uuid "499593d1-a159-11d1-a7e3-00a02444c34e";
  .picture TiPropMethEvent;
  .message[1] Msg1;
  .message[2] Msg2;
  .message[3] Msg3;
  .message[4] Msg4;
  .message[5] Msg5;
  .message[6] Msg6;

  integer I := 123;
  string S  := "Dialog Manager";
  boolean B := true;

  rule void M1
  {
    Info("M1() called.");
    sendevent(this, Msg1);
```

```
}

rule integer M2 (integer I input)
{
  Info("M2(" + I + ") called. Return -123456789");
  return -123456789;
}

rule string M3 (string S input)
{
  Info("M3(" + S + ") called. Return \"Bye\"");
  return "Bye";
}

rule boolean M4 (boolean B input)
{
  Info("M4(" + B + ") called. Return " + ( not B ));
  return ( not B );
}

rule void M5 (integer I input, string S input, boolean B input)
{
  Info("M5(" + I + ", " + S + ", " + B + ") called.");
}

rule string M6 (integer P1 input, string P2 input,
  boolean P3 input, integer P4 input, string P5 input,
  boolean P6 input, integer P7 input, string P8 input)
{
  Info("M6(" + P1 + ", " + P2 + ", " + P3 + ", " + P4 +
       ", " + P5 + ", " + P6 + ", " + P7 + ", " + P8 +
       ") called. Return \"Abracadabra!\"");
  return "Abracadabra!";
}

on .I changed
{
  Info(".I changed := " + this.I);
  this.Gb.Nb.NpProperties.Integer.Value := this.I;
}

on .S changed
{
  Info((".S changed := " + this.S));
  this.Gb.Nb.NpProperties.String.Value := this.S;
}
```

```
on .B changed
{
  Info((".B changed := " + this.B));
  this.Gb.Nb.NpProperties.Boolean.Value := this.B;
}

child groupbox Gb
{
  .xauto 0;
  .yauto 0;
  .borderwidth 0;

  child notebook Nb
  {
    .xauto 0;
    .yauto 1;
    .height 10;

    child notepage NpProperties
    {
      .active true;
      .title "Properties";

      child MInteger Integer { }

      child MString String { .ytop 1; }

      child MBoolean Boolean
      {
        .ytop 2;
        .Value := true;
      }

      child pushbutton PbApply
      {
        .yauto -1;
        .height 1;
        .text "Apply";
        .defbutton true;

        on select
        {
          this.control.I := this.parent.Integer.Value;
          this.control.S := this.parent.String.Value;
          this.control.B := this.parent.Boolean.Value;
        }
      }
```

```
}

child notepage NpMethods
{
  .active false;
  .title "Methods";

  child listbox LbMethods
  {
    .xauto 0;
    .yauto 0;
    .content[1] "void M1()";
    .content[2] "integer M2(integer)";
    .content[3] "string M3(string)";
    .content[4] "boolean M4(boolean)";
    .content[5] "void M5(integer, string, boolean)";
    .content[6] "string M6(integer, string, boolean,   ...\
                 integer, string, boolean, integer, string)";
    .firstchar 1;
  }
}

child notepage NpEvents
{
  .title "Events";

  child listbox LbEvents
  {
    .xauto 1;
    .width 20;
    .yauto 0;
    .ybottom 1;
    .content[1] "Msg1()";
    .content[2] "Msg2(integer)";
    .content[3] "Msg3(string)";
    .content[4] "Msg4(boolean)";
    .content[5] "Msg5(integer, string, boolean)";
    .content[6] "Msg6(integer, string, boolean,        ...\
                      integer, string, boolean)";
    .activeitem 1;
    .firstchar 1;

    on select
    {
      this.parent.Integer.sensitive :=
        (0 <> stringpos(this.content[this.activeitem],
                              "integer"));
```

```
    this.parent.String.sensitive :=
      (0 <> stringpos(this.content[this.activeitem],
                      "string"));
    this.parent.Boolean.sensitive :=
      (0 <> stringpos(this.content[this.activeitem],
                      "boolean"));
  }
}

child MInteger Integer
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .Et.active false;
}

child MString String
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 1;
}

child MBoolean Boolean
{
  .sensitive false;
  .xleft 22;
  .yauto 1;
  .ytop 2;
}

child pushbutton PbSend
{
  .yauto     -1;
  .height    1;
  .text      "Send";
  .defbutton true;
  integer I shadows instance NpEvents.Integer.Value;
  boolean B shadows instance NpEvents.Boolean.Value;
  string  S shadows instance NpEvents.String.Value;

  on select
  {
    case this.parent.LbEvents.activeitem
    in 1:
```

```
            sendevent(this.control, Msg1);
            Info("sendevent(Msg1);");
          in 2:
            sendevent(this.control, Msg2, this.I);
            Info("sendevent(Msg2, " + this.I + ");");
          in 3:
            sendevent(this.control, Msg3, this.S);
            Info("sendevent(Msg3, " + this.S + ");");
          in 4:
            sendevent(this.control, Msg4, this.B);
            Info("sendevent(Msg4, " + this.B + ");");
          in 5:
            sendevent(this.control, Msg5, this.I, this.S,
                      this.B);
            Info("sendevent(Msg5, " + this.I + ", " + this.S +
                           ", " + this.B + ");");
          in 6:
            sendevent(this.control, Msg6, this.I, this.S,
                      this.B, this.I, this.S, this.B);
            Info("sendevent(Msg6, " + this.I + ", " + this.S +
                           ", " + this.B + ", " + this.I +
                           ", " + this.S + ", " + this.B +
                           ");");
          otherwise:
            Info("Error - unknown event");
          endcase
        }
      }
    }
  }

  child listbox LbInfo
  {
    .xauto 0;
    .yauto 0;
    .ytop 10;
    .firstchar 1;
  }
}

on extevent 1
{
  sendevent(this,Msg3,"Bye event");
  this:sendevent(Msg4, true);
}
}
```

```
rule void Info (string S input)
{
  LbInfo.content[(LbInfo.itemcount + 1)] := S;
  LbInfo.topitem := LbInfo.itemcount;
}

on IdmTest extevent 1(object C)
{
  Info("on IdmTest extevent 1");
  sendevent(C, Msg2, 909);
}
```

7 Dialog Manager Environment

The program **idmole** can be used as simulation program for OLE applications. This program has the same options as the simulation program, plus the OLE-specific options described in this manual.

If an application with OLE functionality is to be built, the library **dmole.lib** and **dmoleini.obj** must be linked in addition to the usual libraries.

Other modifications are not necessary.

Index