LBL-33410
UC-405

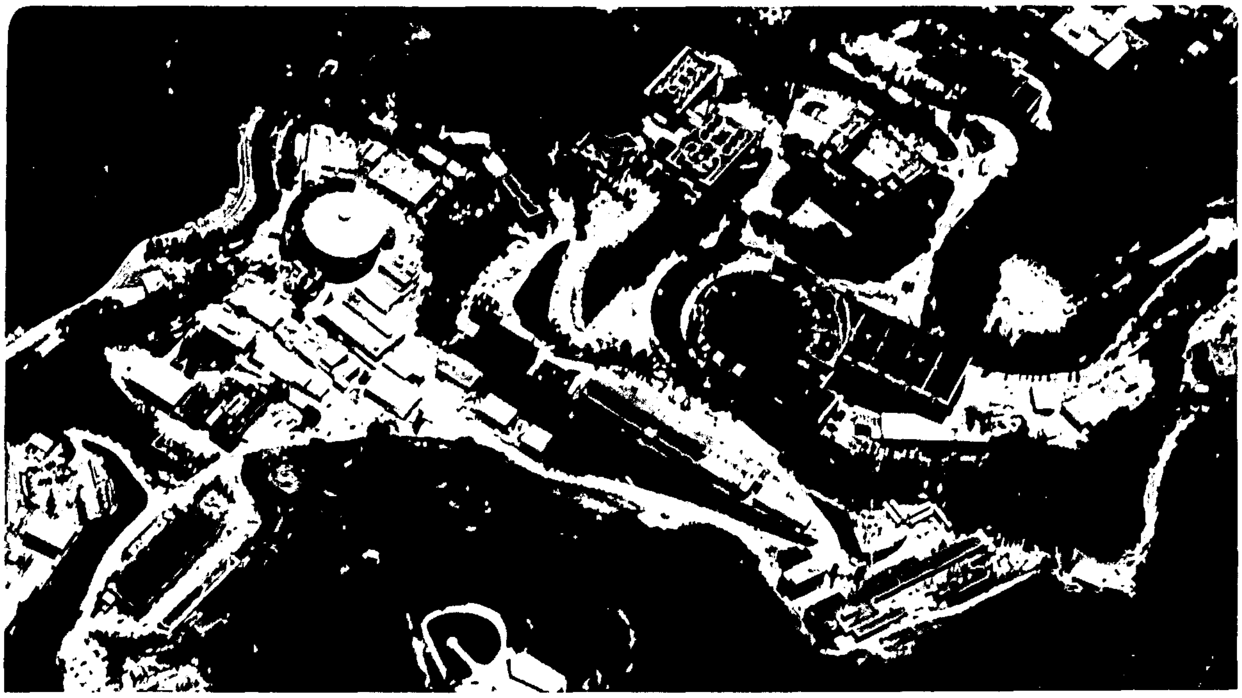
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Information and Computing Sciences Division

OPM Schema Editor 2—A Graphical Editor for Specifying Object-Protocol Structures

I.-M.A. Chen, V.M. Markowitz, F. Pang, and O. Ben-Shachar

July 1993



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

LBL-33410

OPM Schema Editor 2—A Graphical Editor for Specifying Object-Protocol Structures

I-Min A. Chen, Victor M. Markowitz, Francis Pang, and Ofer Ben-Shachar

Information and Computing Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

July 1993

*Issued as Technical Report LBL--33410. This work is supported by the Office of Health and Environmental Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

†Author's e-mail address: ichen@csr.lbl.gov phone: (510) 486-7264, fax: (510) 486-4004
‡Author's e-mail address: VMMarkowitz@lbl.gov phone: (510) 486-6835, fax: (510) 486-4004
**Author's e-mail address: fran@csr.lbl.gov phone: (510) 486-4743, fax: (510) 486-4004
††Author's e-mail address: ofer@netcom.com phone: (415) 325-1214, fax: (415) 322-7470

Contents

Abstract

This document describes an X-window based Schema Editor for the Object-Protocol Model (OPM). OPM is a data model that supports the specification of complex object and protocol classes. Objects and protocols are qualified in OPM by attributes that are defined over (associated with) value classes. Connections of object and protocol classes are expressed in OPM via attributes. OPM supports the specification (expansion) of protocols in terms of alternative and sequences of component (sub) protocols.

The OPM Schema Editor allows specifying, displaying, modifying, and browsing through OPM schemas. The OPM Schema Editor generates an output file that can be used as input to an OPM schema translation tool that maps OPM schemas into definitions for relational database management systems.

The OPM Schema Editor was implemented using C++ and the X11 based Motif toolkit, on Sun SPARCstation under Sun Unix OS 4.1.

This document consists of the following parts:

1. A **tutorial** consisting of seven introductory lessons for the OPM Schema Editor.
2. A **reference manual** describing all the windows and functions of the OPM Schema Editor.
3. An appendix with an **overview** of OPM.

1 Introduction

This document describes the Object-Protocol Model (OPM) Schema Editor, a user-friendly interactive tool for specifying, displaying, modifying, and browsing OPM schemas.

The introduction describes briefly OPM and overviews the OPM Schema Editor. Section 2 contains instructions on starting the OPM Schema Editor. Section 3 contains a tutorial for the OPM Schema Editor. The main window as well as all the dialog windows of the editor are described in detail in Section 4. The OPM data model is described in Appendix A.

1.1 The Object-Protocol Model

The Object-Protocol Model (OPM) is a data model for specifying complex object and protocol structures. Such structures are specific to scientific applications such as molecular biology laboratory information management systems (LIMS). OPM supports the specification of object and protocol classes, object and protocol attributes, class hierarchies, derived attributes, and protocol expansion.

In OPM, an object class is identified by a class name, has a class description, and is associated with attributes that qualify the object class. Attributes take values from value classes that are either other object classes or system provided primitive value classes such as INTEGER or TEXT. For example, an object class CHROMOSOME can have attributes name, map, and owner with value classes CHAR(80), MAP and PERSON, respectively. Attributes can be associated not only with single value classes, but also with union of value classes.

Attributes in OPM can be **simple** or **composite**. A composite attribute consists of multiple component simple attributes. For example, attribute address of class PERSON can be modeled using composite attribute (number, street, city, state, zip_code).

Note that the support for composite attributes and for associating unions of value classes with attributes allows OPM schema designers to avoid the creation of object classes that are artificial, that is, object classes that do not represent entities in the underlying application.

OPM supports the specification of subclass-superclass relationships in an object class (ISA) hierarchy. A subclass is a **specialization** of its superclasses, and inherits all the attributes associated with its superclasses. Multiple inheritance is supported in OPM.

OPM supports the specification of **derived attributes** using derivation rules involving attribute inverse, attribute matching, attribute composition, attribute subvalue, attribute union, arithmetic expressions, and aggregate functions. Attribute inverse and matching provide capabilities for cross referencing values of two or more attributes. For example, let publication be an attribute of object class AUTHOR, and let authors be an attribute of object class PUBLICATION, where publication is associated with value class PUBLICATION, and authors is associated with value class AUTHOR. If publication is specified as the inverse of authors, then for every (value of) publication, say *paper*, of a given AUTHOR, say *John*, the value of attribute authors for PUBLICATION *paper* is *John*.

Protocol classes in OPM are used to model processes such as laboratory protocols. Each instance of a protocol class is an individual experiment. Given an input, a protocol instance (experiment) results in an output, where both input and output consist of objects. OPM supports the recursive specification (expansion) of protocols. **Protocol expansion** in OPM allows specifying a protocol in terms of **alternative** subprotocols, **sequences** of subprotocols, and **optional** subprotocols. A protocol class can be associated with regular as well as **input** and **output** attributes. Input and output attributes are used for specifying input and output connections between protocols. An input (or output) attribute is a regular attribute with additional input (or output) statements indicating its relationship with other input or output attributes. For example, if the result (output) of a CUT protocol is cut_gel, and CUT is followed by a PURITY protocol that takes cut_gel (input) for purifying DNA, then CUT and PURIFY are related via their input and output attributes, that is cut_gel.

OPM has constructs similar to other semantic and object-oriented data models. Thus, in OPM

(1) objects (instances) are classified into object classes and are qualified by attributes that take values from value classes;

(2) object classes are interrelated via attributes and specialization (isa) relationships;

(3) attributes can be defined using various derivation mechanisms, such as inverse and matching.

OPM has two constructs that do not appear in other semantic or object-oriented data models:

(1) the association of attributes with union of value classes;

(2) the definition of protocol classes.

The OPM data model is described in more detail in Appendix A. A full description of OPM can be found in [1].

We intend to implement OPM interfaces on top of relational and object-oriented database management systems (DBMSs). Currently, we develop an OPM interface on top of the Sybase relational DBMS. For relational DBMSs such as Sybase, we use the *Extended Entity-Relationship* (EER) model as an intermediate level between OPM and the underlying relational DBMS. Thus, we map OPM schemas into EER schemas and queries, and subsequently map EER schemas and queries into relational database schema definitions and SQL queries using existing EER to DBMS translation tools [3, 4]. The mapping of OPM schemas into EER schemas and queries is described in [2].

1.2 The OPM Schema Editor

The OPM Schema Editor is used to specify, display, modify, and browse OPM schemas. An OPM schema generally consists of objects and protocol classes. Each class is associated with attributes.

The main menu of the OPM Schema Editor provides commands to create a new schema, load an existing schema, save a current schema to a file and to invoke dialog windows for defining or modifying OPM classes and attributes.

The OPM Schema Editor starts by default a new schema. If an existing schema is needed, then **Open** menu item must be used in order to load the schema. The current definition of the OPM schema can be viewed via the editor windows. The **New** menu item resets the editor for a new schema. In order to save the current schema, **Save** or **Save As** menu items can be used.

In order to add new object classes to the current schema, **Define OPM Object Class** menu item is used. A new (empty) **Object Class Definition Window** (see Figure 4) will pop up for defining a new object class. **Define OPM Protocol Class** or **Define Controlled Value Class** are used to add a new protocol class or a controlled value class, respectively.

The editor supports the definition of the following main OPM meta entities:

1.Object Class,

2.Protocol Class,

3.Controlled Value Class,

4.Simple Attribute (for an object or a protocol class),

5.Composite Attribute (for an object or a protocol class), and

6.Input/Output Attribute (for a protocol class).

The OPM Editor supplies a dialog window to define, display and modify each of the constructs above. Each dialog window has buttons (such as **New**, **Clear** or **Help**) that invoke different actions or functions. It is important to note that invoking **New** or **Modify** commands in a window associated with one of the main OPM meta entities, entails changing the current internal definition of the schema. When a **Modify** command is invoked in another window of the editor, only the content of one of the main OPM meta entities is changed without changing the current definition of the schema; the schema will be changed only when the **New**, **Modify** or **Delete** button on the main OPM meta entity is invoked. Thus, the state of each of the OPM meta entities is always reflected by the dialog window that represents it.

Print OPM in Latex and **Print PostScript** menu items output the current schema definition in

OPM schema definition language to a Latex file and a PostScript file, respectively.

Quit menu item allows leaving the editor. If there are schema changes that have not been saved, the user will be required to confirm the quit action.

2 Starting the OPM Schema Editor

The current version of the OPM Schema Editor can be run on a Sun SPARCstation running Sun OS 4.1 (or above) and X-Window R11.2 (or above). It is recommended to run the editor using the Motif window manager (mwm). The editor supports all the standard X toolkit command line options plus one of its own: **nobell** or **nobells** turn warning and error beeps off.

The editor requires the following four files: the file containing the editor executable code, the X application defaults file, the UID file, and the configuration file. The editor executable code is called **editor**, the X application defaults file is called **SchemaEditor**, the **UID** file is called **Editor.uid**, and the configuration file is called **metadb.i**. The **editor, Editor.uid**, and **metadb.i** files should be in the current working directory. However, if you have the **metadb.i** and **Editor.uid** files stored in, for example / home/editor directory, you can use the following command to specify them to the **editor**:

```
setenv EDITOR_UID /home/editor/Editor.uid
setenv EDITOR_META /home/editor/metadb.i
```

The directory where **SchemaEditor** is installed is indicated using the following command:

```
setenv XAPPLRESDIR <directory>
```

If the **SchemaEditor** is installed at a certain location in the system, such as: /usr/lib/X11/app-defaults/ SchemaEditor, then it is loaded automatically and this step is not required.

In order to run the editor, first load the default environment with the following command

```
xrdb -load SchemaEditor
```

next, start the editor by typing

```
editor
```

then click **Continue** on the copyright notice window.

For new users we recommend first the tutorial, a step-by-step, hands-on introduction to the editor and its features (see next section).

3 Tutorial

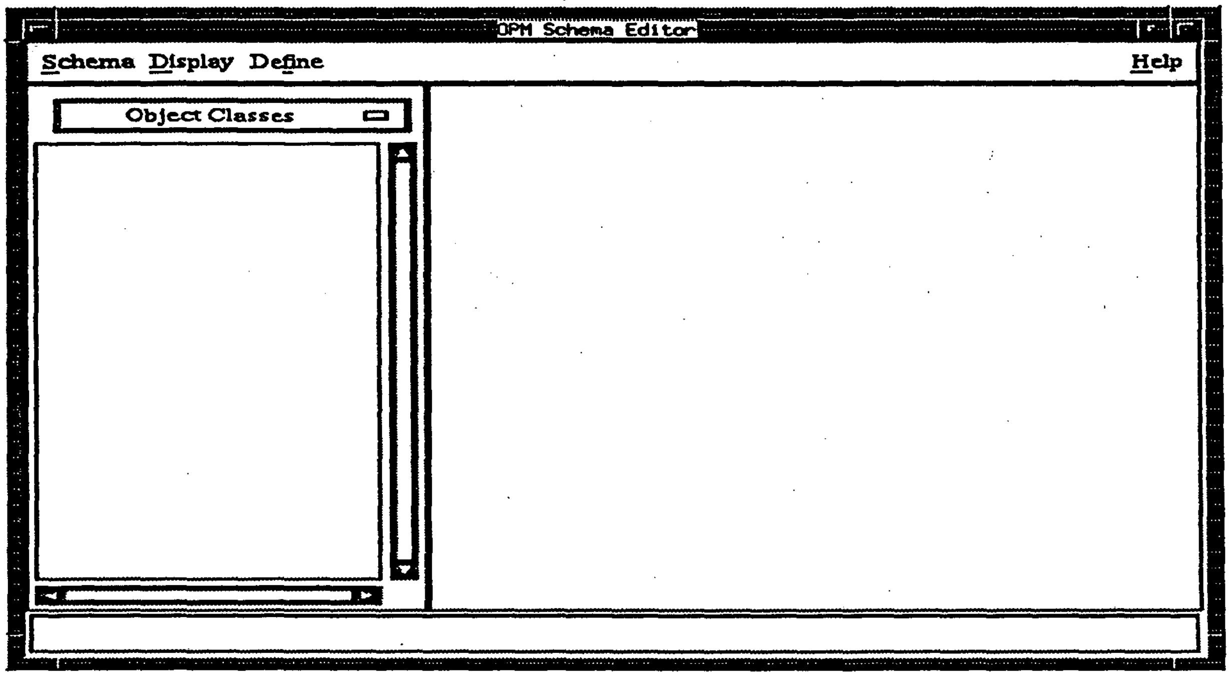
This section contains a tutorial designed to help learning the OPM Schema Editor. The first part of the tutorial presents a predefined schema that represents a brief outline of the editor. The second part of the tutorial is a guided step-by-step specification of a schema for a simple database. This part consists of six lessons, each providing step-by-step instructions for completing the tasks.

3.1 Basic Editor Usage

3.1.1 Starting the Editor

The files required to run the editor as well as the command starting the editor are described in the previous section. For running this part of the tutorial an additional file, called **Tutorial.OPM** is needed. For simplicity, put this file in the current working directory.

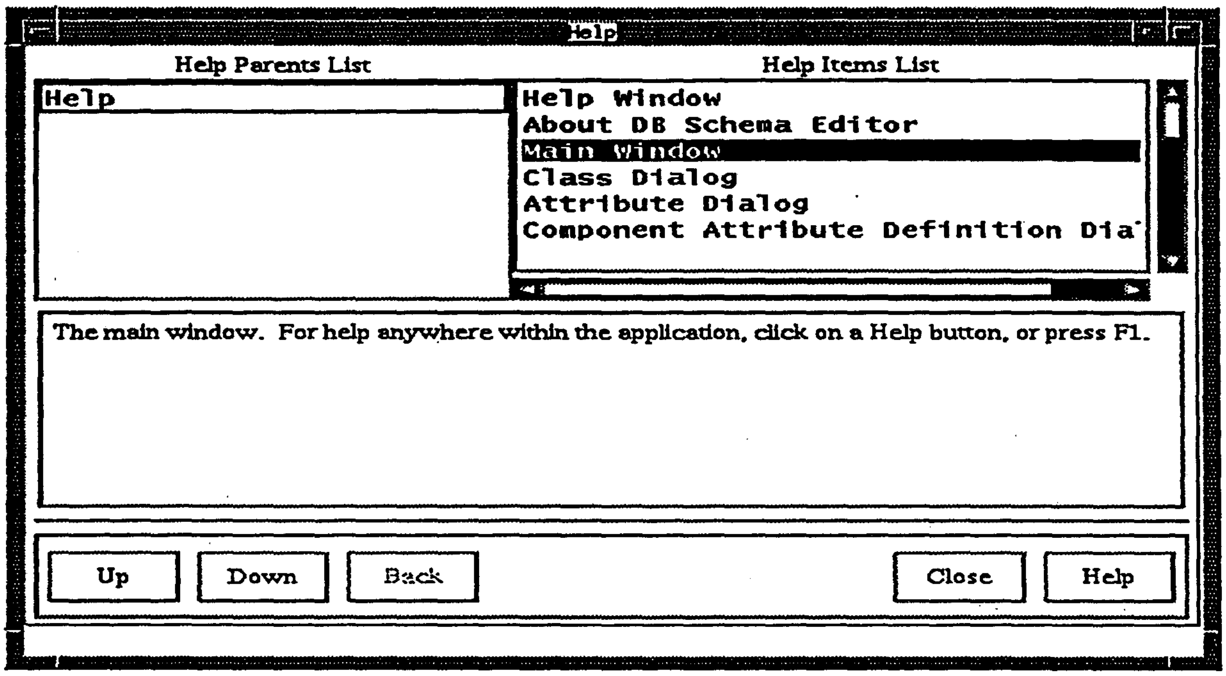
After starting the editor and clicking on the **Continue** button of the copyright notice, the editor's main window is brought up.



The main window contains the main menu bar across the top, the classes listbox and its associated option menu at the left, and the main window drawing area in the remainder of the window.

The first thing you can explore is the help tree: from the **Help** menu, select **Help**. The **Help** window displays the topic of the main window. Help about other windows can be obtained by clicking on the topics listed in the **Help Items List** Listbox.

For details on a topic (i.e., on its sub-topics) click **Down**; to return to a higher level, click **Up**.



The main window. For help anywhere within the application, click on a Help button, or press F1.

3.1.2 Looking at an Existing Schema

This section illustrates the schema specification process of the OPM Schema Editor.

Select **Open** from the **Schema** menu in the main menu bar. All the OPM schema files used by the editor are assumed to have file extension **.OPM**. Therefore, a standard Motif file selection dialog box listing all the files ***.OPM** in its **Files** Listbox will appear. Double clicking on **Tutorial.OPM** loads the file into the editor.

Note that the **Object Classes** Listbox contains several entries: this is an alphabetically ordered list of the object classes defined in this schema. The list of the protocol classes in this schema can be displayed by selecting **Protocol Classes** from the option menu heading the listbox. Similarly, the list of controlled value classes in this schema can be displayed by selecting **Controlled Value Classes** from the option menu heading the listbox.

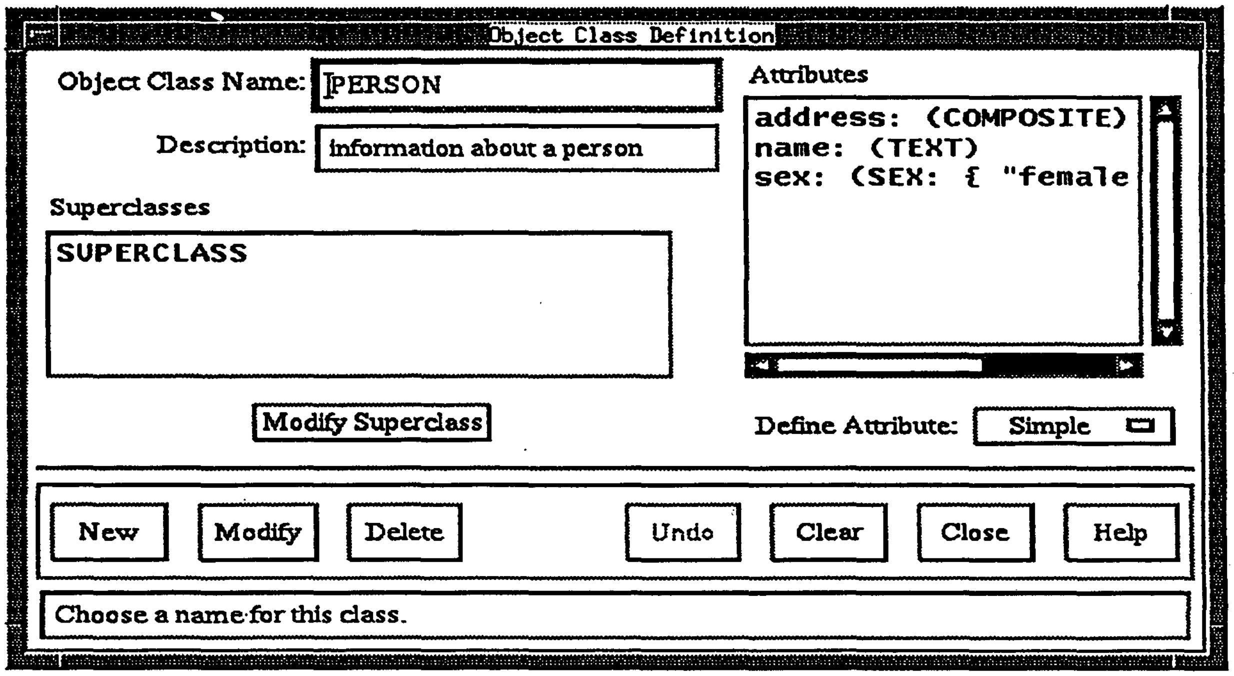
Display the list of object classes and click on PERSON. A graphical representation of the PERSON object class and its superclass is displayed in the main window drawing area. Now switch to **Protocol Classes** in the option menu heading the listbox, and click on CREATE_SCHEMA_FILE. For protocol classes such as CREATE_SCHEMA_FILE, a graphical representation of their expansion (i.e., their decomposition into alternative or sequences of protocol steps), is shown (if specified, of course). Expansions are explained in more detail later.

There are three modes for the graphical display in the main window. The default mode, which you are seeing now, is called **Class Links** mode. Bring down the **Display** menu from the main menu bar and switch to **Class Hierarchy** mode. Notice that now a complete hierarchy tree of the OPM object classes is displayed. Switch to **Detailed Links** mode. This looks very much like **Class Links** mode.

Double clicking on any of the buttons displayed in the main window in any of the graphical display modes will open the class definition window for that class. This is an alternative to double clicking in the main window listbox.

In **Class Links** mode or **Detailed Links** mode, single clicking on any button that represents a class, for example clicking on the SAVE_FILE button in the expansion diagram of CREATE_SCHEMA_FILE, will cause the graphical representation of that class to expand, that is, to replace the button. Clicking again will reverse this expansion. The expansion of the graphical display is limited to a maximum of six levels.

Unlike in **Class Links** mode, in **Detailed Links** mode (sub)protocols that are expanded inside the expansion of another protocol are displayed without their attributes. **Class Links** mode therefore provides a more concise graphical representation.



Object Class Name: PERSON
Description: information about a person
Superclasses: SUPERCLASS
Attributes:

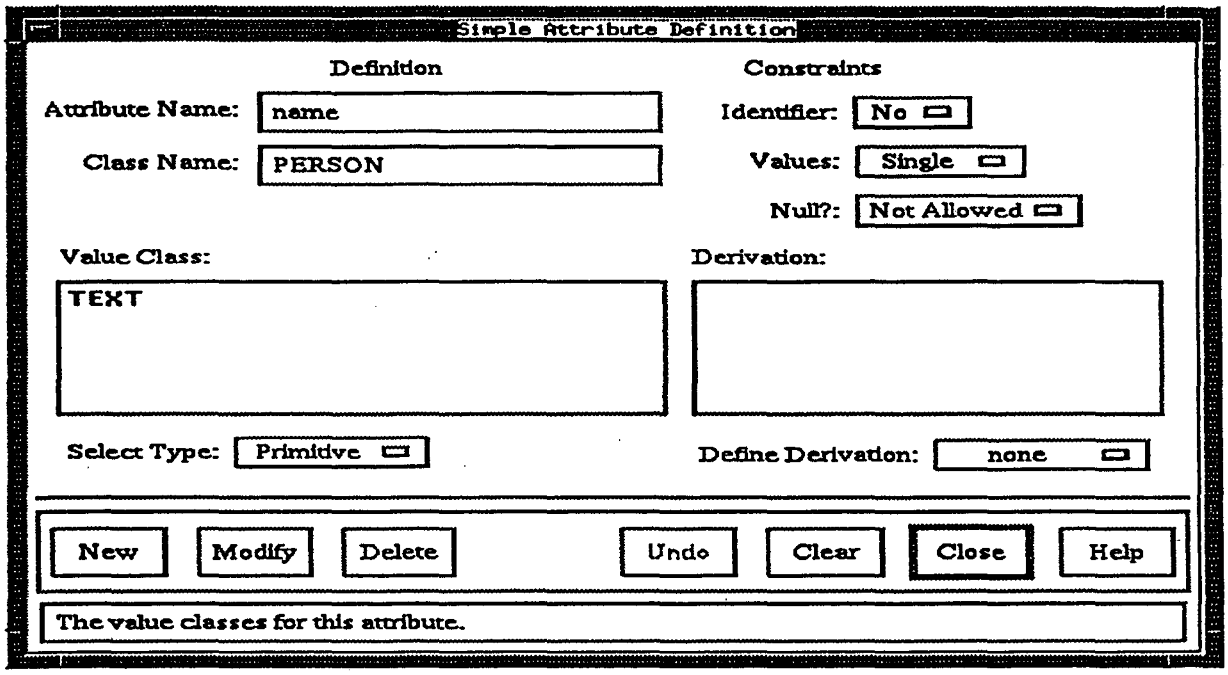
```
address: (COMPOSITE)
name: (TEXT)
sex: (SEX: { "female
```

Define Attribute: Simple

Choose a name for this class.

Return now to the list of object classes. You can look at the details of the PERSON class by double clicking on its name in the listbox. The editor opens the **Object Class Definition** window.This window displays the name of the class, a description, its superclasses, and its attributes.

Look at the name attribute of object class PERSON by double clicking it in the **Attributes** Listbox of the **Object Class Definition** window. Attribute name is a simple attribute, and a **Simple Attribute Definition** window is brought up.

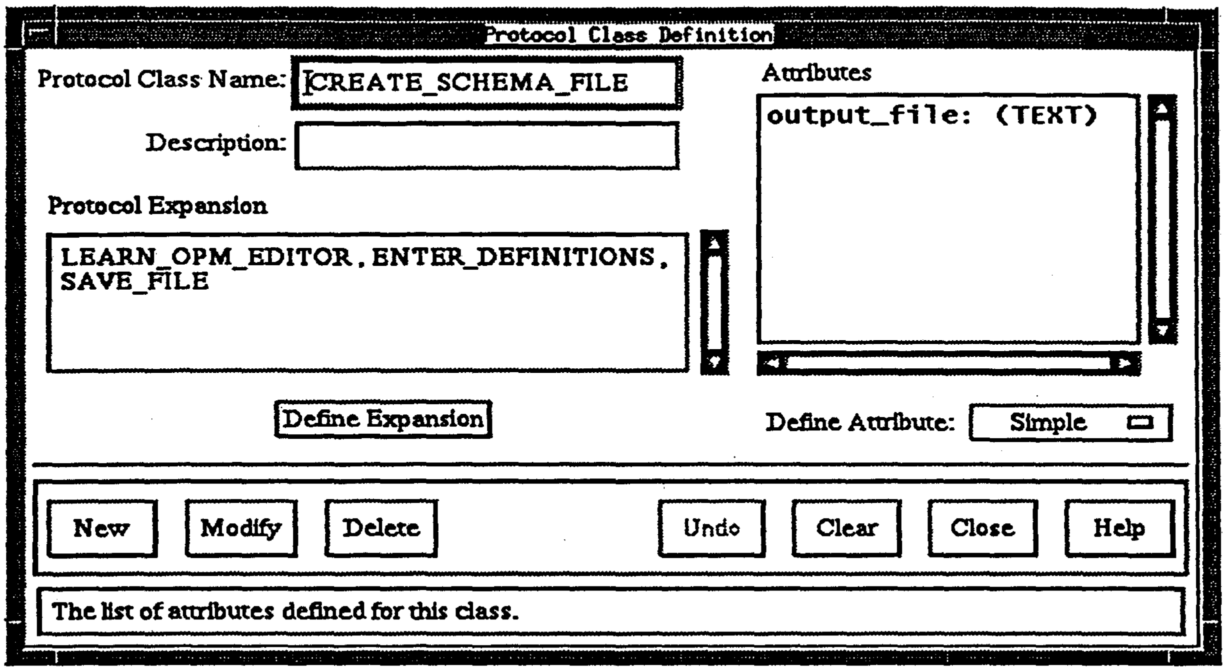


The value classes for this attribute.

The attribute name and the name of the class for this attribute are displayed at the top of the window. The value class of name is listed in the **Value Class** Listbox (it is primitive value class TEXT). The attribute constraints displayed in the upper right part show that name is not an **Identifier** attribute, it is **Single** valued, it is not allowed to have **Null** values. The **Derivation** Listbox is empty because name does not haye a derivation.

If you have tried to modify something in these windows and want to close them, a warning message indicates that your work is not saved. Go ahead and close them anyway, up to the main window. In the main window, switch to the **Protocol Classes** option for the listbox. Double click on the CREATE_SCHEMA_FILE protocol.

Protocol Class Definition window is very similar to the **Object Class Definition** window. Protocol classes do not have superclasses, but have protocol expansions displayed in the **Protocol Expansion** area. This window has an expansion expressing the fact that the CREATE_SCHEMA_FILE protocol consists of three sub-protocols (steps), LEARN_OPM_EDITOR followed by ENTER_DEFINITIONS and SAVE_FILE. Close the CREATE_SCHEMA_FILE **Protocol Class Definition** window.



CREATE_SCHEMA_FILE

LEARN_OPM_EDITOR, ENTER_DEFINITIONS, SAVE_FILE

output_file: (TEXT)

The list of attributes defined for this class.

In the main window, display the controlled value classes by switching to the **Controlled Value Classes** option of the main window **Object Classes** Listbox. Double click on the first entry.

The **Controlled Value Class/Value** window displays a controlled value class called SEX, consisting of two values: *male* and *female*. The Value Type of both *male* and *female* is Character String. Any time an attribute can take values from a finite set of predefined (controlled) values, its value class can be defined as a controlled value class. Close this window.

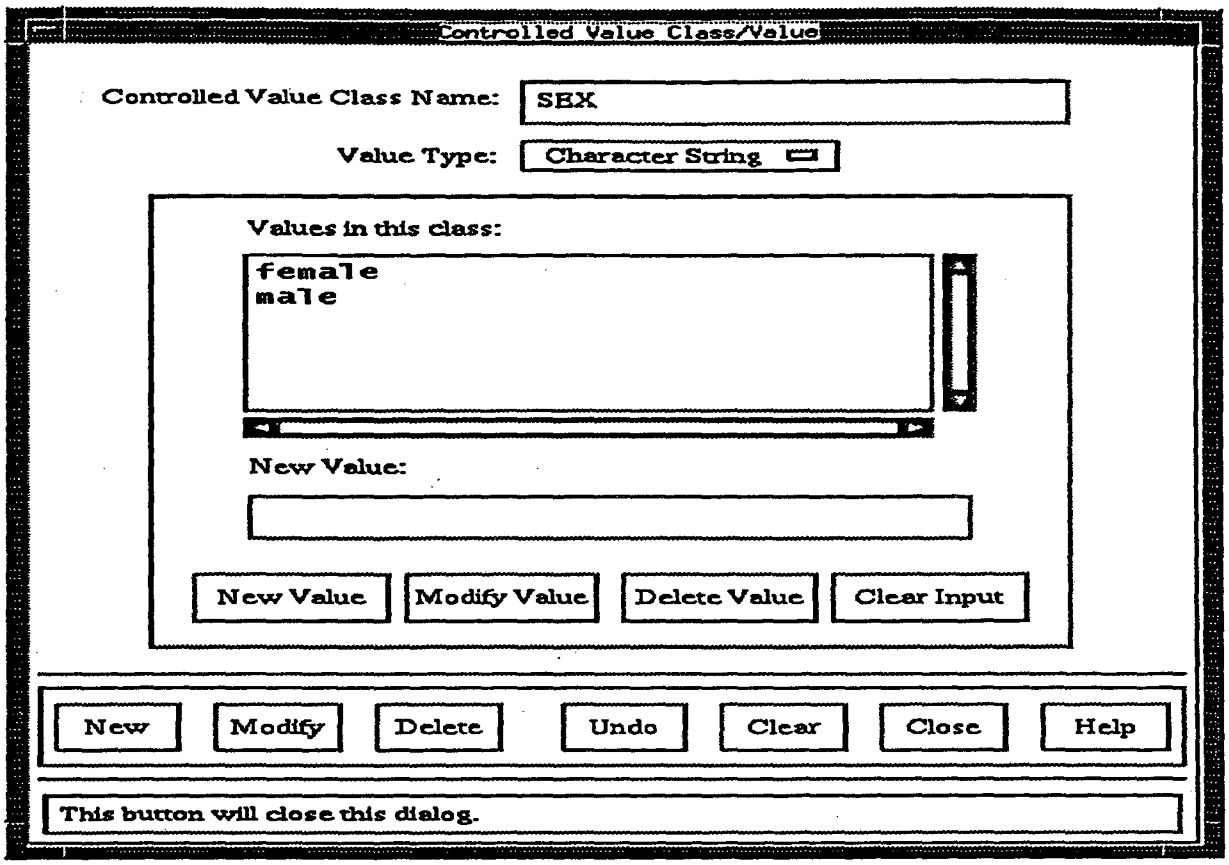
From the **Schema** menu in the main menu bar, select New. This selection clears the editor (i.e., removes the current schema, if any) and allows specifying a new schema from scratch. You are ready for the second part of the tutorial.

3.2 Specifying and Saving Schemas

This part of the tutorial guides you through a step-by-step specification of a schema for a simple database. Suppose that you need to fill a position in your group.You will specify the schema of a database that can be used by interviewers, keeping track of people, resumes, and the steps in the interview process. The tutorial consists six lessons regarding the specification of:

1. object classes and protocol classes;

2. attributes for object and protocol classes;



Controlled Value Class Name: SEX

Value Type: Character String

female
male

This button will close this dialog.

3. controlled value classes, and associating attributes with value classes;

4. derivation expressions for attributes;

5. expansions for protocol classes;

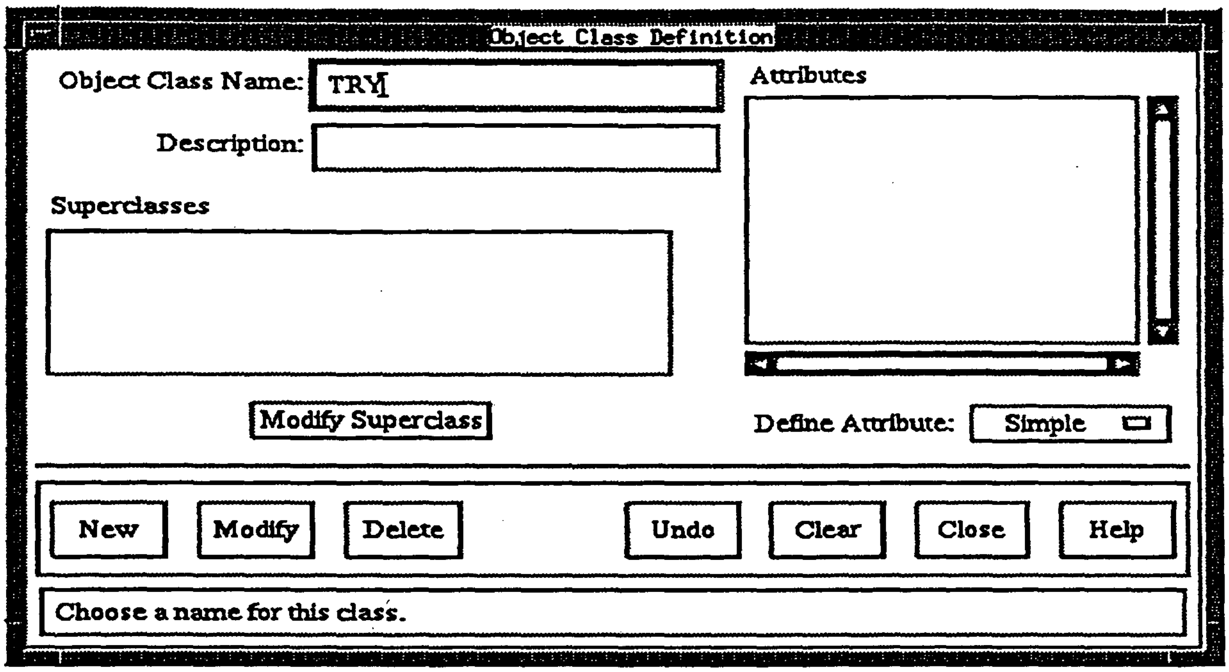
6. connections for input/output attributes of protocol classes.

Each stage of the schema specification process is based on the previous ones, so the lessons should be done in order. It is worth saving from time to time partially specified schemas.

3.2.1 Specifying Object Classes

In order to define an object class, select **OPM Object Class** from the **Define** menu in the main menu bar. An empty **Object Class Definition** window is brought up.

Call this class TRY by typing the name in the **Object Class Name** area. Note that you cannot type in lower-case characters. Class names are always in upper-case characters, and the window automatically converts the characters into upper-case. Type a short description in the **Description** area, and click



Object Class Name: TRY

Define Attribute: Simple

Choose a name for this class.

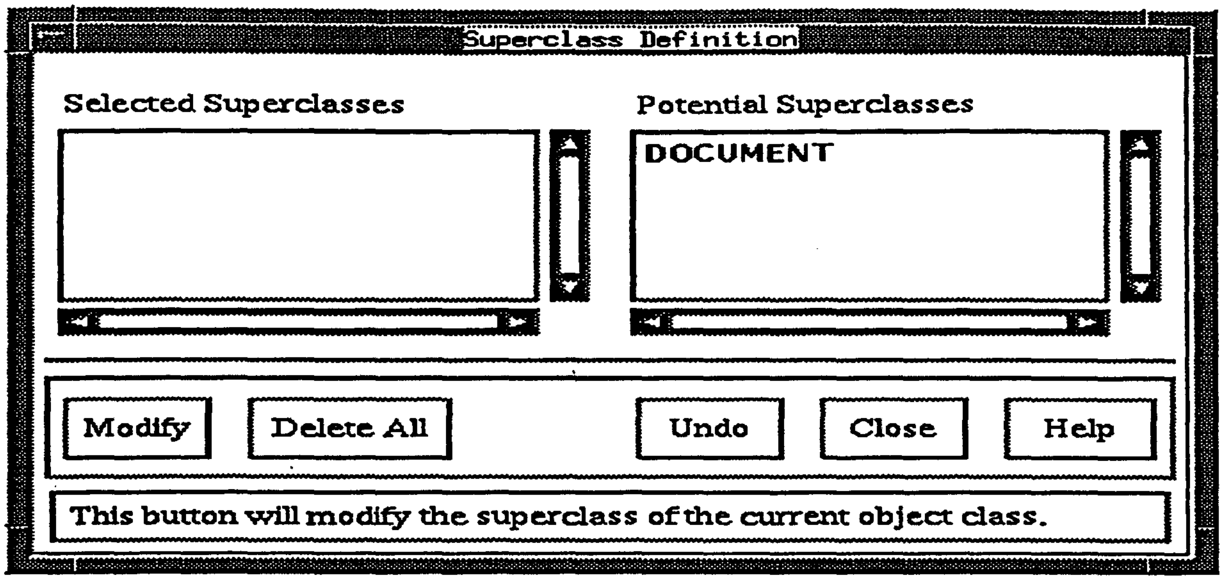
the **New** button. This new object class has been added to your schema. The schema will consist of more than one object class, so you may want to change the name of this class. In the **Object Class Name** area backspace over TRY, and type DOCUMENT. Try clicking **Close** to close this window. The name change has not been recorded, therefore a warning dialog box is brought up. Click **Cancel** in the warning dialog box, then click **Modify** to change the name of the class.

If you want to reuse a class definition, then you can use a previously defined class and change the name. Starting with the DOCUMENT class definition, change the name to RESUME and click **New**. Now you have two classes.

Click the **Modify Superclass** button for defining a superclass for RESUME.

The **Superclass Definition** window shows that only DOCUMENT is a potential superclass for RESUME, because it is the only other class defined so far. Click on DOCUMENT in the **Potential Superclasses** Listbox. DOCUMENT is moved from this listbox to the **Selected Superclasses** Listbox. Click **Modify** in the **Superclass Definition** window. Note that the newly defined superclass is now displayed in the **Superclasses** Listbox of the **Object Class Definition** window. Click **Close** in the **Superclass Definition** window.

Before proceeding with the schema specification, hit **F1** on the keyboard. This is the help key; the **Help** window is brought up. The topic shown is the **Superclass** field of the **Object Class Definition** window. Anywhere in the application, if you are not sure how to use a button or what is the function of



This button will modify the superclass of the current object class.

a menu, etc., you can click **F1** when keyboard focus is on the widget in question. The help is context sensitive and will display the appropriate topic. Click **Modify** in the **Object Class Definition** window, to save the newly defined superclass, and close this window.

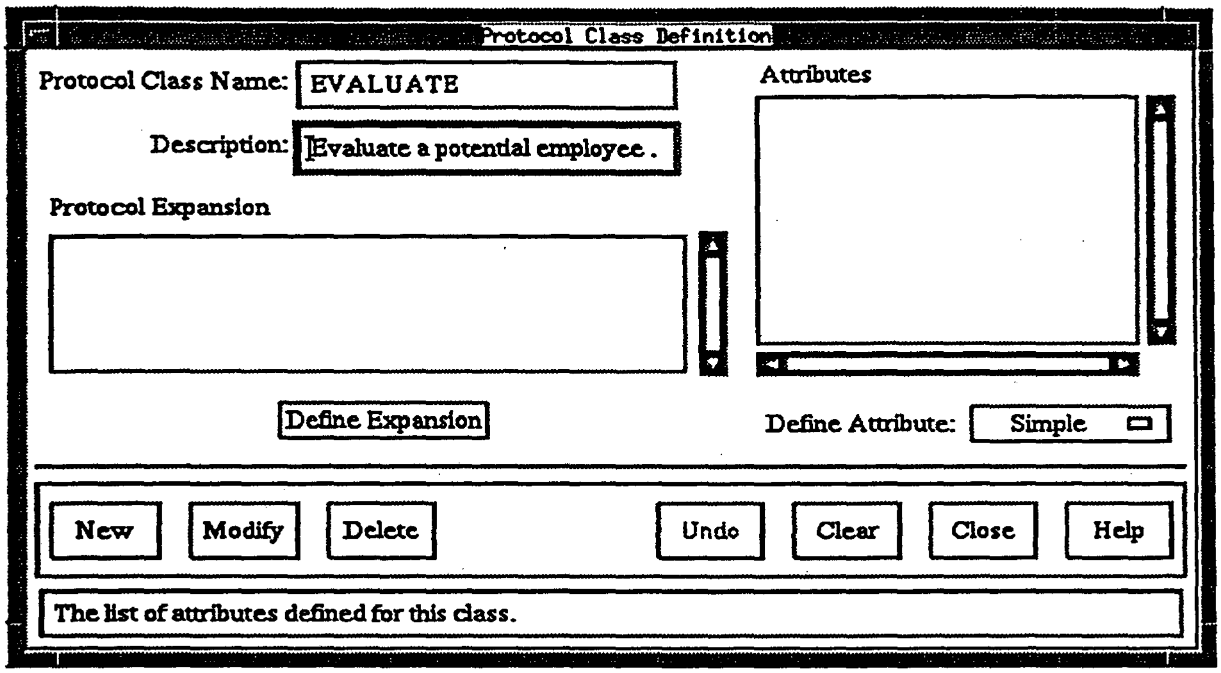
3.2.2 Specifying Protocol Classes

Protocol classes are used to keep track of procedures and processes. For this database, you may want to keep track of interviews and phone calls. Bring up the **Protocol Class Definition** window by selecting **OPM Protocol Class** from the **Define** menu in the main menu bar. An empty **Protocol Class Definition** window is brought up.

Call this protocol class EVALUATE by entering this name into the **Protocol Class Name** area. Enter a short description(e.g. "Evaluate a potential employee.") in the **Description** area. Click **New** to add this protocol class to database schema. Clear the window by clicking **Clear**, and enter a new protocol class name, TELEPHONE. Click **New**. Clear the window again and enter INTERVIEW. Click **New**. Clear the window once more and enter HIRE. Click **New**. Three more protocol classes have been added to the schema.

You then realize that hiring a successful candidate is done only for one person, so it is not necessary to keep track of the hiring in the database. To delete the HIRE protocol class, click the **Delete** button. A warning is issued: "Are you sure you want to delete the current class?". You confirm the deletion by clicking **OK**, and the class is deleted.

You will define an expansion for a protocol class later. Now it is worth saving your work. From the



Protocol Class Name: EVALUATE

Description: Evaluate a potential employee.

The list of attributes defined for this class.

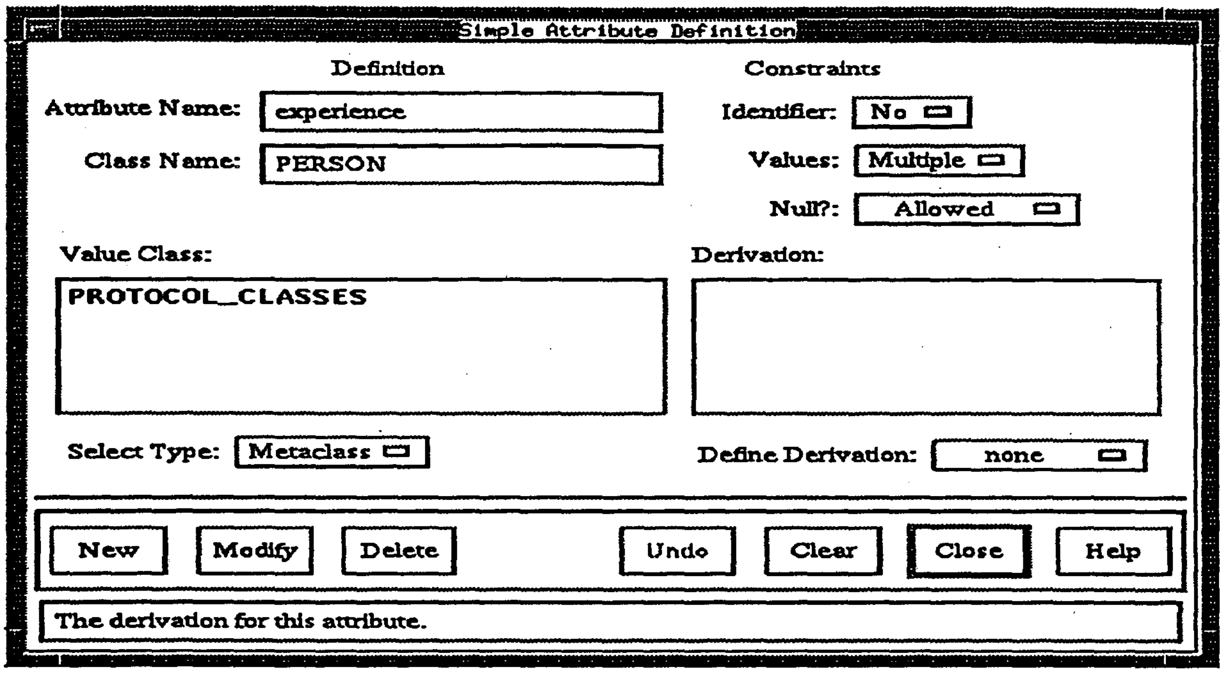
Schema menu, select **Save As**. The file selection listbox allows you to choose a name for your new schema. Remember to use a file name with extension **.OPM** such as **Example.OPM** or **Employee. OPM**. Now that you have chosen a name for the schema, using **Save** instead of **Save As** will save subsequent schema versions to the same file.

3.3 Specifying Attributes

3.3.1 Specifying Simple Attributes

Open an empty **Object Class Definition** window from the main menu. Specify an object class called PERSON by typing this name in **Object Class Name** area. Click **New** to add this object class. In order to define a simple attribute for a PERSON, click **Simple** in the **Define Attribute** option menu. A **Simple Attribute Definition** window is brought up.

Call this attribute experience by typing this name in the **Attribute Name** area. A person can have lots of job experience, so this attribute should be specified as multi-valued. Select **Multiple** from the **Values** option menu. If you click **New** now, you will get an error. Attributes cannot be created without a value class. You do not know what the value class is for this attribute, so you can try choosing a very general value class. Select **Metaclass** from the value class **Select Type** option menu. In the **Attribute Metaclass Value Class** window there are two choices: OBJECT_CLASSES and PROTOCOL_CLASSES



Attribute Name: experience
Class Name: PERSON
Identifier: No
Values: Multiple
Null?: Allowed
Value Class: PROTOCOL_CLASSES
Derivation:
Select Type: Metaclass
Define Derivation: none
The derivation for this attribute.

(besides Undefined). Job experience refers to performing tasks in general, so let us choose PROTOCOL_-CLASSES. Click **Modify** and close the **Attribute Metaclass Value Class** window. Now click **New** in order to add this attribute to the schema.

Note that the attribute and its value class are displayed and highlighted now in the **Attributes** Listbox of the **Object Class Definition** window. The highlighting indicates that this is the selected (current) attribute. If you wish to delete or modify an attribute now, this is the one that would be affected.

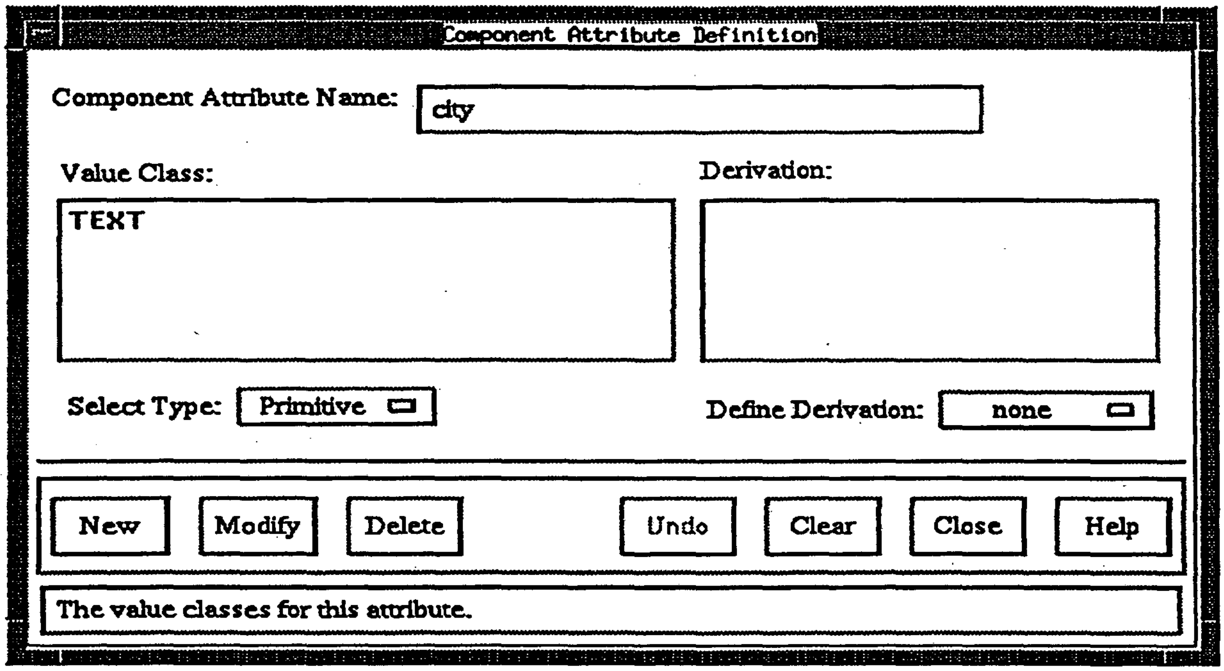
3.3.2 Specifying Composite Attributes

The persons interviewed have addresses, and addresses are composite rather than simple attributes. Select **Composite** from the **Define Attribute** option menu in PERSON **Object Class Definition** window. Call this attribute address by typing it in the **Attribute Name** area.

Now we show how to add the new component attribute street_address. After you click the **Define Component** button, a new **Component Attribute Definition** window is brought up.

Type in the attribute name in **Component Attribute Name** area in this window. This attribute is associated with primitive value class TEXT. Click **Primitive** in **Select Type** option menu. Select TEXT in **Primitive Classes** Listbox in **Attribute Primitive Value Class** window, click **Modify,** and then close this window. Click **New** in the **Component Attribute Definition** window to add this component attribute.

Change the attribute name in **Component Attribute Definition** window to city and click **New** . A second component with value class TEXT has been defined.



Component Attribute Name: city

TEXT

Select Type: Primitive

Define Derivation: none

The value classes for this attribute.

Clear the component window by clicking **Clear**, enter a new name state and associate the attribute with primitive value class CHAR(n). In the **Length** area of the **Attribute Primitive Value Class** window enter the length for the CHAR(n) - in this case, 2. Click **Modify**, and state now is associated with value class CHAR(2). Click New again in the **Component Attribute Definition** window. Close the **Component Attribute Definition** window.

Remember to click **New** in the **Composite Attribute Definition** window to add the new composite attribute to object class PERSON.

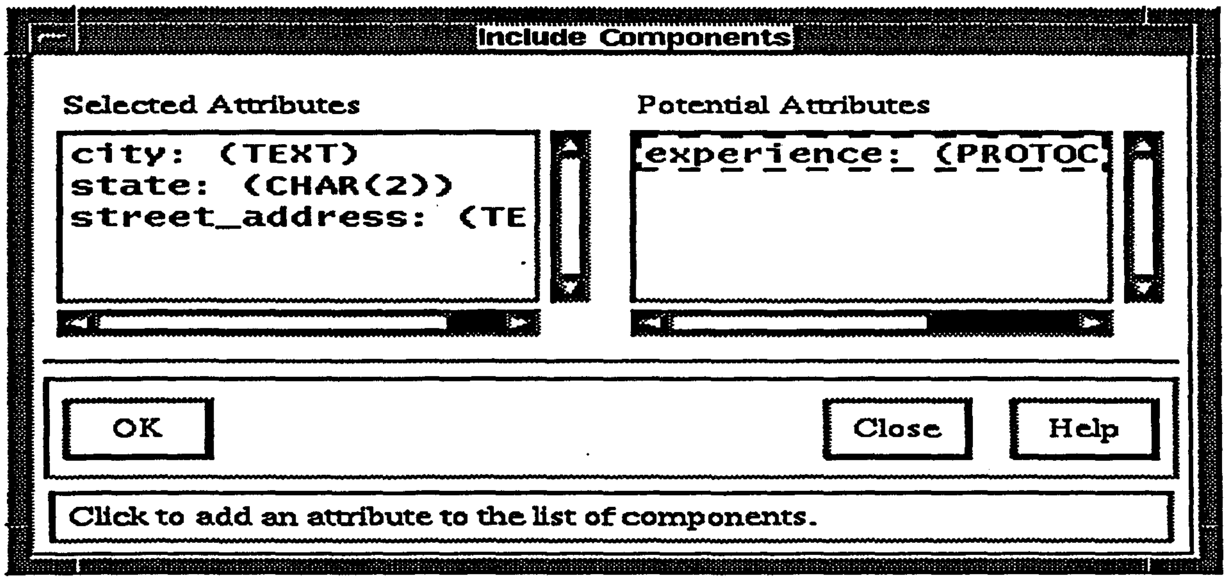
3.3.3 Composing and Decomposing Attributes

A composite attribute, such as the address attribute you have just created, can be broken into simple attributes. Display the composite attribute address by double clicking on its name in the **Attributes** listbox of the PERSON **Object Class Definition** window.

Click on the button **Decompose**. As a result, the composite attribute address is replaced by its components, street_address, city, and state, that are now simple attributes of the PERSON class. Close the composite attribute window and verify this by looking in the **Attributes** listbox. There is no composite attribute named address, but there are new simple attributes named street_address, city, and state.

The reverse action is also very easy to carry out. Instead of specifying components for a composite attribute one by one, a composite attribute can be defined by selecting its components from the list of existing simple attributes.

In order to restore the composite attribute address, open the **Composite Attribute** window and type the name address in the name field. Open the **Include Components** window by clicking on the **Include Components** button below the **Components** Listbox.



```
city: (TEXT)
state: (CHAR(2))
street_address: (TE
```

```
experience: (PROTOC
```

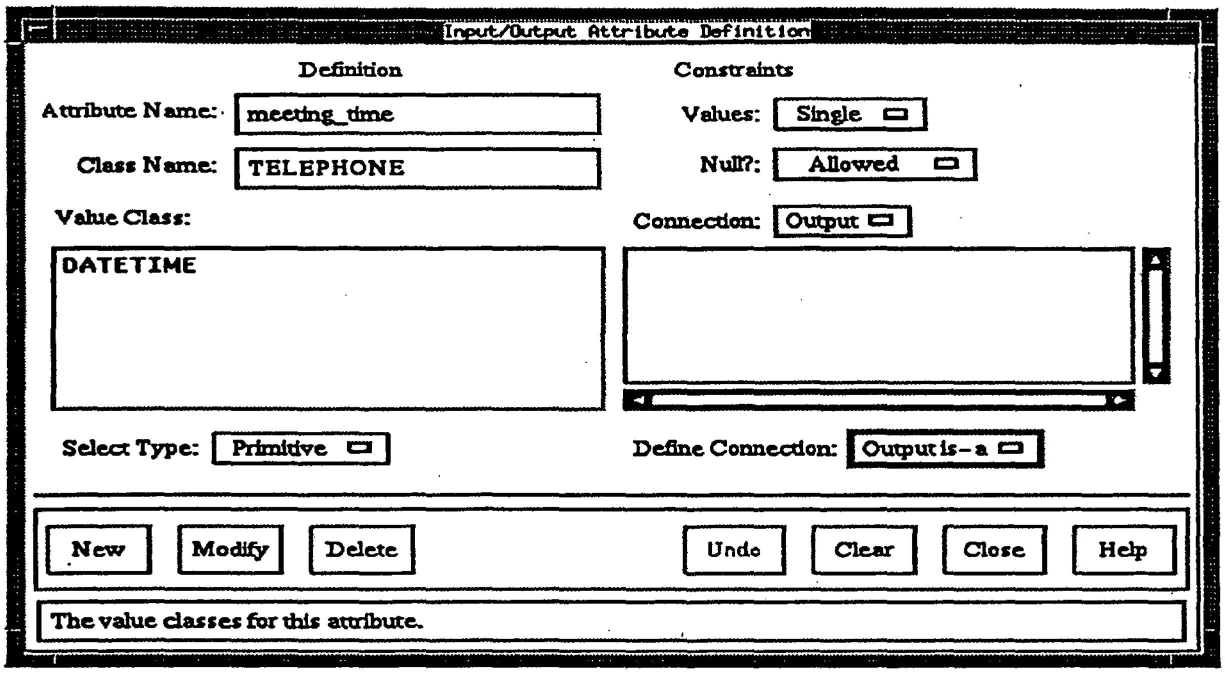
Click to add an attribute to the list of components.

The **Include Components** window allows you to select the attributes that will be moved from the list of attributes to the list of component attributes for this composite attribute. Select street_address, city, and state by clicking on them in the **Potential Attributes** Listbox. Confirm these selections by clicking on the **OK** button. The **Include Components** window will be closed and the **Composite Attribute** window will show that you have selected three new components. Click the **New** button now in order to create the address attribute.

Close the **Composite Attribute Definition** window and verify the changes in the **Attributes** listbox of the **Object Class Definition** window. PERSON class now has a new attribute called address, while street_address, city, and state attributes have been removed.

3.3.4 Specifying Input/Output Attributes

Only protocol classes can have input or output attributes. Double click on TELEPHONE to bring up the **Protocol Class Definition** window. Select **Input/Output** in the **Define Attribute** option menu, and the **Input/Output Attribute Definition** window is brought up.


Input/Output Attribute Definition

Definition

Attribute Name: meeting_time

Class Name: TELEPHONE

Value Class:

DATETIME

Select Type: Primitive

Constraints

Values: Single

Null?: Allowed

Connection: Output

Define Connection: Output is-a

The value classes for this attribute.

Let us specify an attribute called meeting_time that represents a meeting time set up by telephone for an interview. Bring up the **Attribute Primitive Value Class** window, select DATETIME, click **Modify**, and then **Close**. The attribute has a name and a value class. This represents the *result* of an action (telephone call), so it is an *output* attribute. Select **Output** in the **Connection** option menu. Click **New** in order to associate this attribute with the TELEPHONE protocol class.

3.4 Specifying Value Classes

Four types of value classes are supported by the OPM Schema Editor: *controlled*, *primitive*, *abstract*, and *metaclass*. Several primitive value classes and a metaclass have been already associated with attributes up to now. The other two types of value classes are explained below.

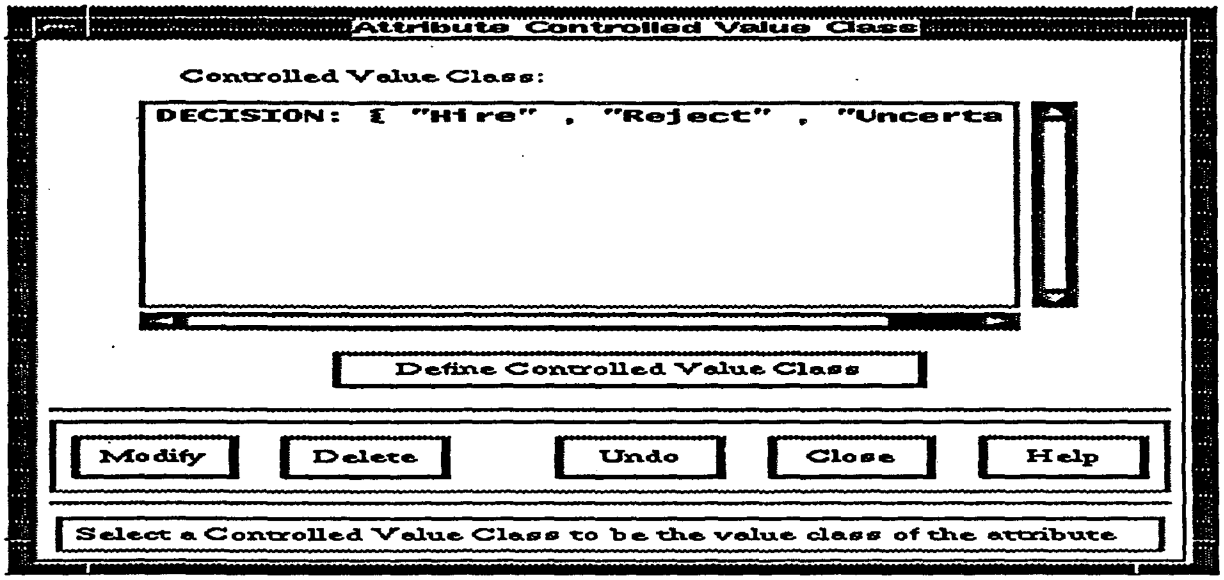
3.4.1 Specifying Controlled Value Class

You need to keep track of the interview results. An interview can result in hiring the candidate immediately, reject the candidate immediately, or postpone the decision (perhaps set up another interview). Open the INTERVIEW **Protocol Class Definition** window, and specify an input/output attribute called decision. Select **Controlled** from the value class **Select Type** option menu.

An empty **Attribute Controlled Value Class** window is brought up, because there are no controlled value classes defined for this schema.

Click on the **Define Controlled Value Class** button to bring up the **Controlled Value Class/Value** window; this window can also be brought up from the main menu bar, by selecting **Controlled Value Class** under the **Define** menu.

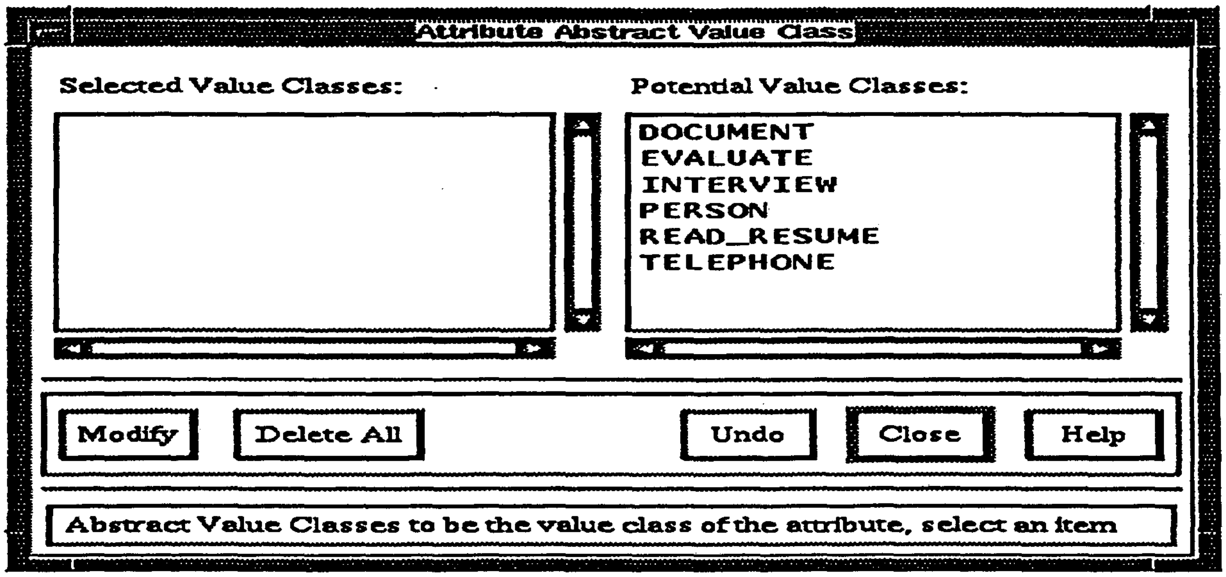
Specify a controlled value class called DECISION by entering the name of the value class in the **Controlled Value Class Name** area at the top of the window. The first (controlled) value for DECISION, Hire, must be entered in the **New Value** area. Click the **New Value** button; Hire appears in the **Values in this class** Listbox. Enter Reject in the **New Value** area and click **New Value** again. Similarly, enter value Uncertain. Click the New button and then close the window. The new controlled value class DECISION is added to the schema.


Attribute Controlled Value Class
Controlled Value Class:
DECISION: { "Hire" , "Reject" , "Uncerta
Define Controlled Value Class
Modify
Delete
Undo
Close
Help
Select a Controlled Value Class to be the value class of the attribute

This new controlled value class is listed in the **Controlled Value Class** Listbox in the **Attribute Controlled Value Class** window. In order to associate this value class with attribute decision, select the DECISION in the listbox, and then click **Modify**. Close the **Attribute Controlled Value Class** window. In the **Input/Output Attribute Definition** window, the new value class is now displayed. Select the **Output** option for **Connection** and click New in order to associate this attribute with the INTERVIEW protocol class.

3.4.2 Specifying Abstract Value Classes

Clear the **Protocol Class Definition** window and specify a new protocol class called READ_RESUME. Click **New** in order to add this class to the schema. Open the **Input/Output Attribute Definition** window, and specify input attribute resume; specify this attribute so that it is not allowed to have null values. The value class of this new attribute should be RESUME, of course. Such a value class (defined as an OPM class) is called *abstract*. Select **Abstract** from the value class **Select Type** option menu. The **Attribute Abstract Value Class** window is brought up.



Abstract Value Classes to be the value class of the attribute, select an item

The **Potential Value Classes** Listbox contains all the classes that have been defined so far. Select RESUME from this listbox. RESUME is moved from the **Potential Value Classes** Listbox to the **Selected Value Classes** Listbox. Click **Modify** and **Close** in this window. Click **New** in the **Input/Output Attribute Definition** window in order to associate the attribute with its protocol class.

An attribute can have more than one abstract value class. For example, suppose that a candidate has as reference a recommendation from a previous manager or co-worker. This reference could be in the form of a person to contact or in the form of a recorded reference letter. Open the PERSON object class. Open the **Simple Attribute Definition** window to add a simple multi-valued attribute reference. Open the **Attribute Abstract Value Class** window, and select two object classes DOCUMENT and PERSON. Click **Modify**. A reference attribute can be now either a DOCUMENT or a PERSON.

Note that an attribute does not need to be multi-valued in order to be associated with several value classes. For example, a candidate can have only one reference, but the reference can be either a letter or a person.

Troubleshooting: If you ever see the **Abstract Value Class Definition** window completely empty when you are trying to indicate an abstract value class, then you have not defined any classes yet. This window will be empty as long as there are no defined classes.

3.5 Specifying Attribute Derivations

Derived attributes are associated with object or protocol classes, and are derived from other attribute(s) using a derivation rule. An attribute can be associated with at most one derivation rule. Each type of derivation has constraints and is defined in its own separate window. The seven types of derivations rules are:

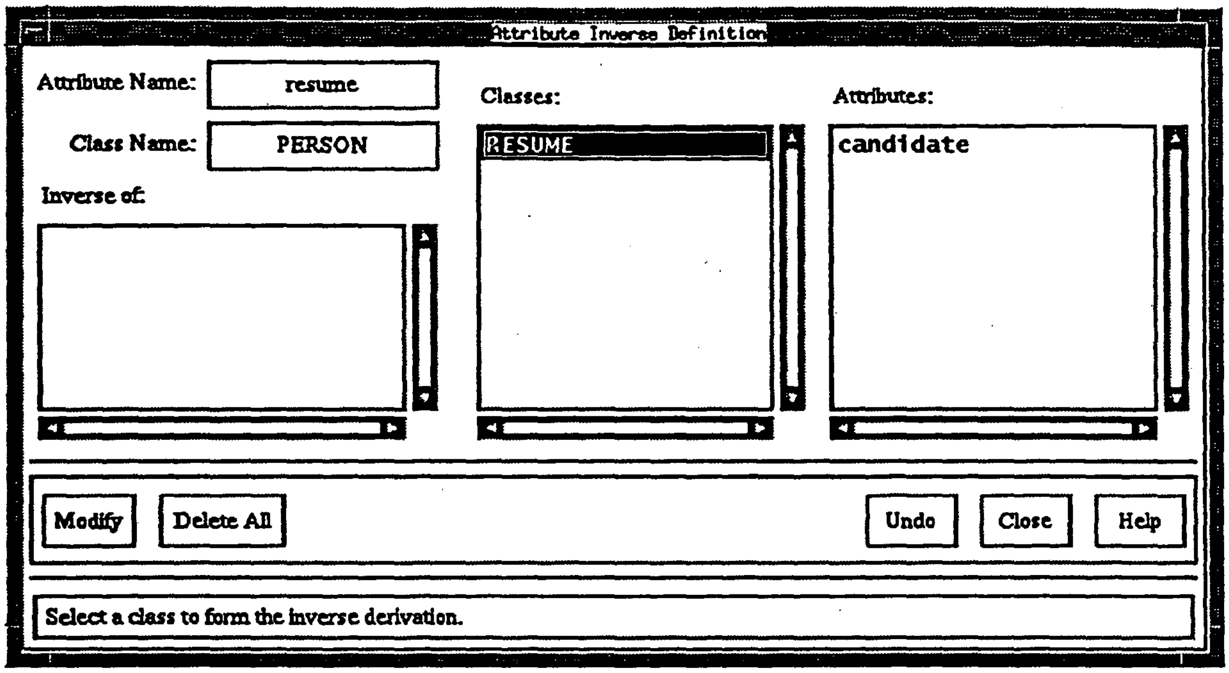
1. **inverse**: the derived attribute is the inverse of an attribute associated with another object or protocol class;
2. **match**: the derived attribute matches an attribute of another object or protocol class on a component attribute;
3. **arithmetic**: the derived attribute is computed from an arithmetic expression involving arithmetic operators, constants, and other numeric attributes of the same object or protocol class;
4. **aggregate**: the derived attribute is computed by applying an aggregate function on a numeric attribute of the same object or protocol class, or by counting the values of another attribute of the same object or protocol class;
5. **composition**: the derived attribute is a composition of other attributes;
6. **subvalue**: the derived attribute is defined as a subvalue of another attribute from the same class;
7. **union**: the derived attribute is defined as the union of other attributes from the same class.

A derived attribute cannot be an identifier, nor an input or output attribute of a protocol class. Composite derived attributes are allowed in OPM only using attribute matching.

3.5.1 Specifying Attribute Inverse Derivations

Create a simple attribute candidate for object class RESUME, and associate this attribute with abstract value class PERSON. Also create a simple attribute resume for object class PERSON; associate it with abstract value class RESUME. After clicking **New** to add the attribute, do not close the **Simple Attribute Definition** window. Instead, select **inverse** in the **Define Derivation** option menu. The **Attribute Inverse Definition** window is brought up.

Because attribute resume is associated with only one value class RESUME, its inverse attribute must be associated with RESUME. Conversely, the potential inverse attribute must be associated with value class PERSON. RESUME is listed in the **Classes** Listbox in **Attribute Inverse Definition** window. The RESUME object class has only one attribute, candidate, whose value class is PERSON and is listed in the



resume

PERSON

RESUME

candidate

Select a class to form the inverse derivation.

Attributes Listbox.

Select candidate in the **Attributes** Listbox. The new inverse definition, RESUME.candidate, is listed in the **Inverse of** Listbox. Click **Modify** in order to update the attribute definition.

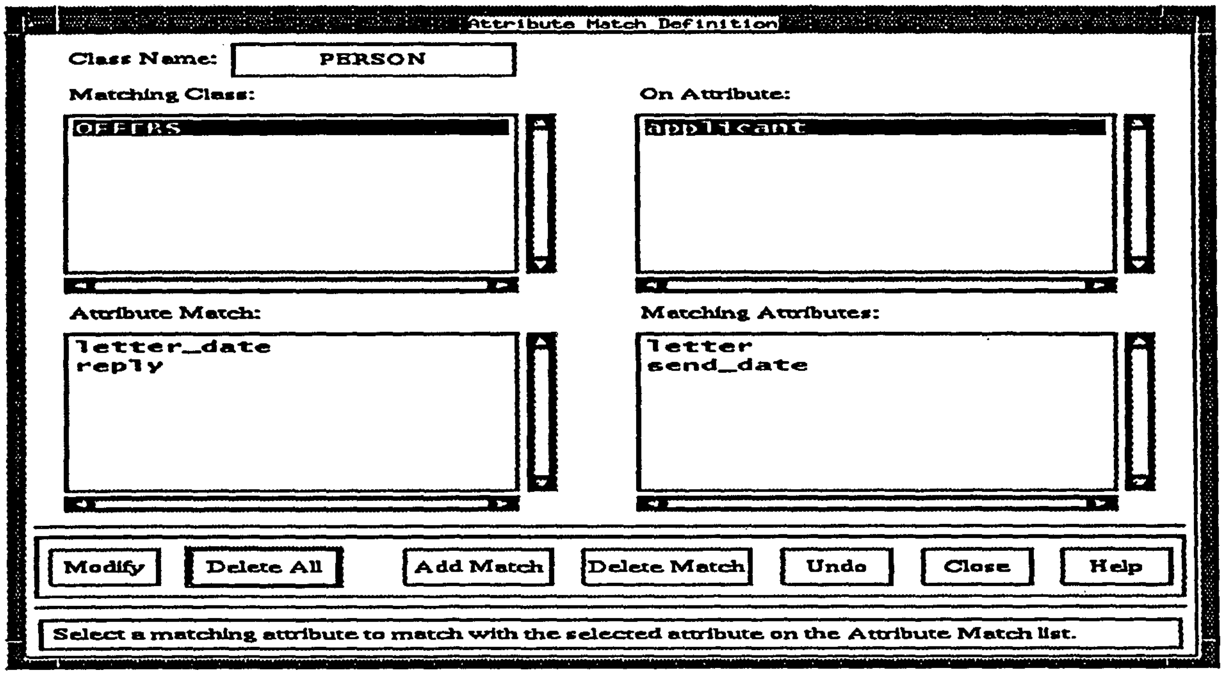
Inverse specifications can be mutual, therefore the editor asks if you want to specify PERSON. resume as an inverse of attribute candidate of RESUME . Select **Yes** , and close the **Attribute Inverse Definition** window. The inverse derivation is displayed in the **Derivation** area in the **Simple Attribute Definition** window.

Note that a simple or component attribute associated with an abstract value class can have an inverse derivation, provided the attribute is not specified as an identifier.

3.5.2 Specifying Attribute Match Derivations

In the simplest case of matching, a simple attribute A of object or protocol class C_1 can match an attribute B of object or protocol class C_2 on attribute M only if (B, M) is defined as a composite attribute of C_2, A and B have identical value classes, and the value class of M includes C_1.

Suppose that there are several positions available, and the positions are offered to all qualified applicants by sending them letters on different dates. Furthermore, every applicant receiving an offer letter records the letter date and the letter. First, create a new object class called LETTER without attributes. Create a new object class called OFFERS associated with a composite attribute consisting of three components: applicant, associated with abstract value class PERSON, send_date, associated with primitive value class DATETIME, and letter, associated with abstract value class LETTER. Save the OFFERS object class, and then associate object class PERSON with a new composite attribute consisting of two components: letter_date, associated with primitive value class DATETIME, and reply, associated with abstract value class LETTER. Click **New** on the **Composite Attribute Definition** window to make the change. Select **match** in the **Define Derivation** option menu in order to bring up the **Attribute Match Definition** window.



Select a matching attribute to match with the selected attribute on the Attribute Match list.

In the **Attribute Match Definition** window, the **Matching Class** Listbox lists only object class OFFERS, since only OFFERS is qualified to be involved in an attribute matching with an attribute of object class PERSON. The **Attribute Match** Listbox lists components (letter_date and reply) of the new composite attribute of PERSON; they are used to match component attributes of OFFERS.

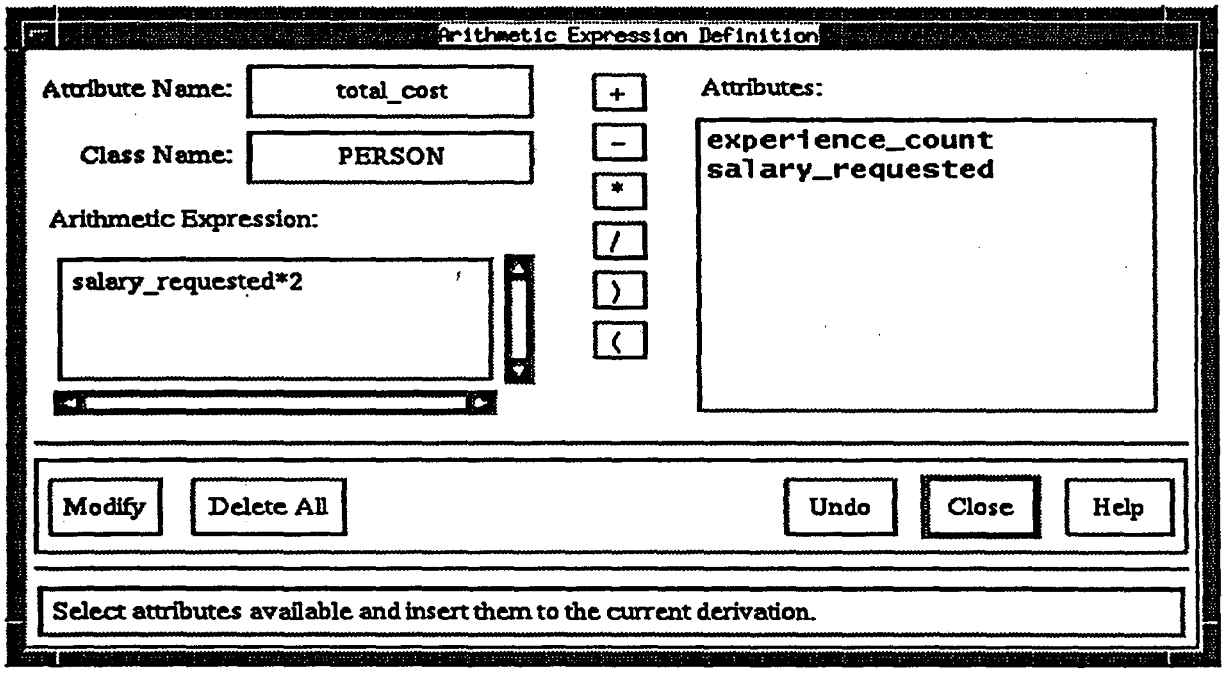
Select OFFERS in the **Matching Class** Listbox. Component attribute applicant appears in the **On Attribute** Listbox. After selecting applicant in this listbox, the other two components of the same composite attribute appear in the **Matching Attributes** Listbox: they are send_date and letter.

Select letter_date in the **Attribute Match** Listbox and send_date in the **Matching Attributes** Listbox, and then click the **Add Match** button in order to match these two component attributes. Select reply in the **Attribute Match** Listbox and then letter in the **Matching Attributes** Listbox, and then click the **Add Match** button again. Click the **Modify** button in order to record this match derivation for the composite attribute of PERSON. The matching expression appears in the **Derivation** area in **Composite Attribute Definition** window. Click **Modify** button on the **Composite Attribute Definition** window in order to update schema.

Note that only non-identifier simple or composite attributes can have a match derivation.

3.5.3 Specifying Arithmetic Expression Derivations

An arithmetic derivation defines the value of a numerical attribute in terms of the values of other numerical attributes of the same class. For example, an applicant can request a certain salary. In order to represent such a request, a new simple attribute called salary_requested is created for object class PERSON; this attribute is associated with primitive value class MONEY.


Arithmetic Expression Definition

Attribute Name: total_cost

Class Name: PERSON

Arithmetic Expression:

salary_requested*2

Attributes:

experience_count
salary_requested

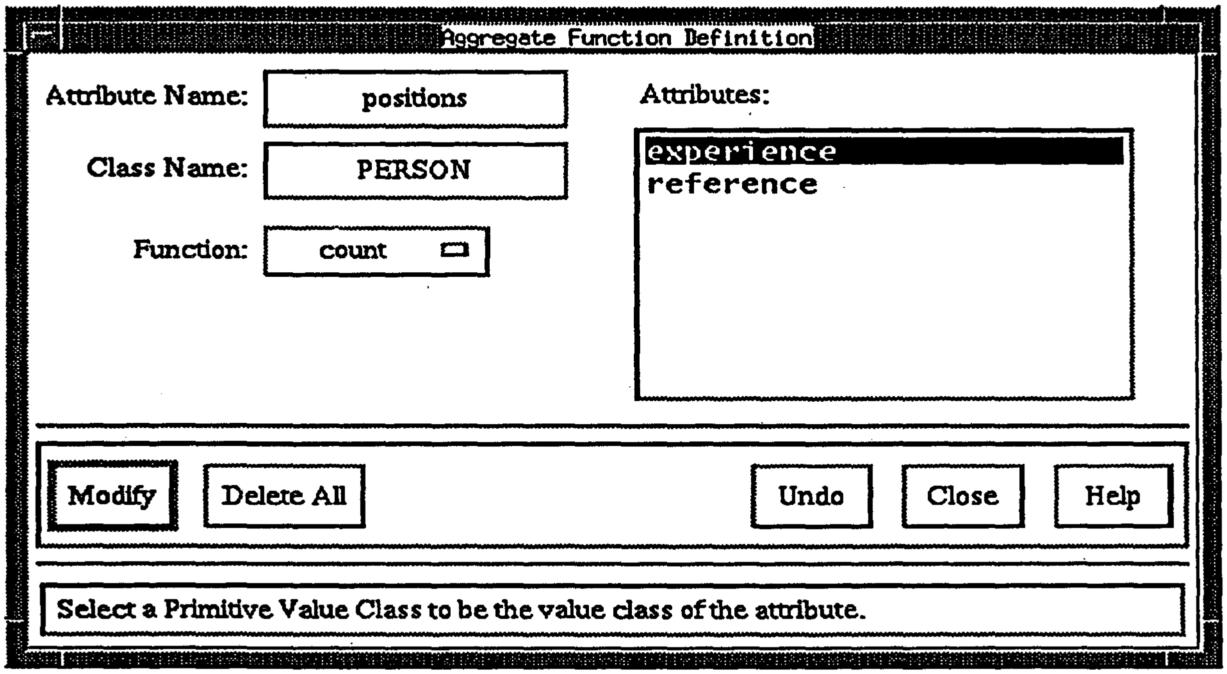
Select attributes available and insert them to the current derivation.

The cost of a person, however, includes, in addition to the salary, benefits and overhead. Suppose that on the average, an employee costs twice her/his salary. Create a new simple attribute called total_cost for PERSON with value class MONEY. Click **New** to add this attribute. In the **Simple Attribute Definition** window, select **arithmetic** in the **Define Derivation** option menu. The **Arithmetic Expression Definition** window is brought up.

Specify the arithmetic expression salary_requested * 2 by first selecting salary_requested in **Attributes** Listbox, then clicking on the * function button, and finally by editing in the **Arithmetic Expression** area in order to add number 2. All this can be also done by typing directly the expression in the **Arithmetic Expression** area. Click **Modify** in order to update the definition of attribute total_cost. The derivation is displayed in the **Derivation** area in the **Simple Attribute Definition** window. Remember to click **Modify** in order to save the change. Only simple attributes with *numerical* primitive value classes INTEGER, SMALLINT, REAL, FLOAT, or MONEY can have arithmetic derivations, and such attributes cannot be specified as identifiers.

3.5.4 Specifying Aggregate Function Derivations

Similar to arithmetic derivations, aggregate function derivations define the value of a numerical attribute in terms of other attributes of the same class. However, an aggregate derivation involves only one multi-valued attribute that is not restricted to have numerical value classes. For example, an applicant can have job experience from several positions. The number of positions held by an applicant can be represented using an attribute that counts the number of different values associated with (multi-valued) attribute experience.



Attribute Name: positions
Class Name: PERSON
Function: count
Attributes:
experience
reference
Select a Primitive Value Class to be the value class of the attribute.

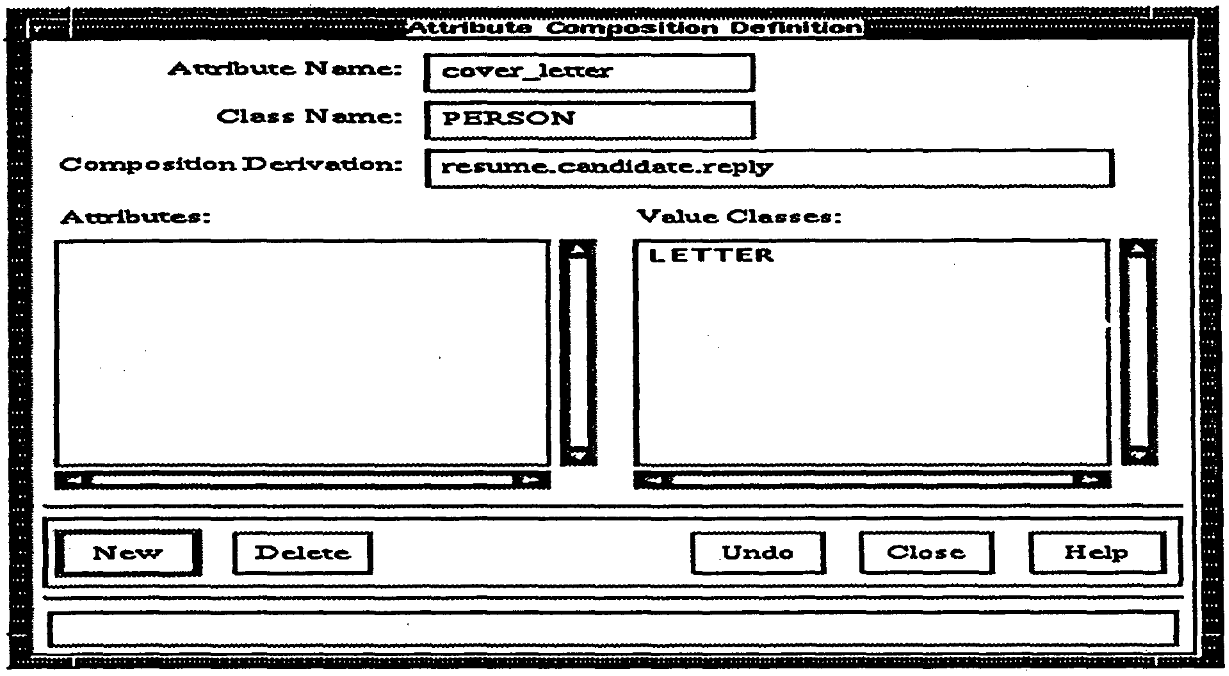
Create a new simple attribute for PERSON called positions, associated with primitive value class SMALLINT. After clicking **New** in order to specify this attribute, open the **Aggregate Function Definition** window by selecting **aggregate** in the **Define Derivation** option menu. The **Aggregate Function Definition** window is brought up.

The **Function** option menu allows choosing one of several aggregate functions. Select the **count** aggregate function. The **Attributes** Listbox lists all the multi-valued attributes of PERSON that could be involved in this derivation. Select experience from the listbox. Click **Modify**, and then close this window. The new aggregate function derivation is displayed in the **Derivation** area in the **Simple Attribute Definition** window. Remember to click **Modify** in this window in order to associate the attribute definition with the new derivation expression.

An attribute associated with an aggregate function derivation cannot be an identifier.

3.5.5 Specifying Attribute Composition Derivations

Create a simple attribute cover_letter for the object class PERSON, and associate this attribute with abstract value class LETTER. Click the **New** button on the **Simple Attribute Definition** window and select **composition** in the **Define Derivation** option menu. The **Attribute Composition Definition** window is brought up.


Attribute Composition Definition

Attribute Name: cover_letter

Class Name: PERSON

Composition Derivation: resume.candidate.reply

Value Classes: LETTER

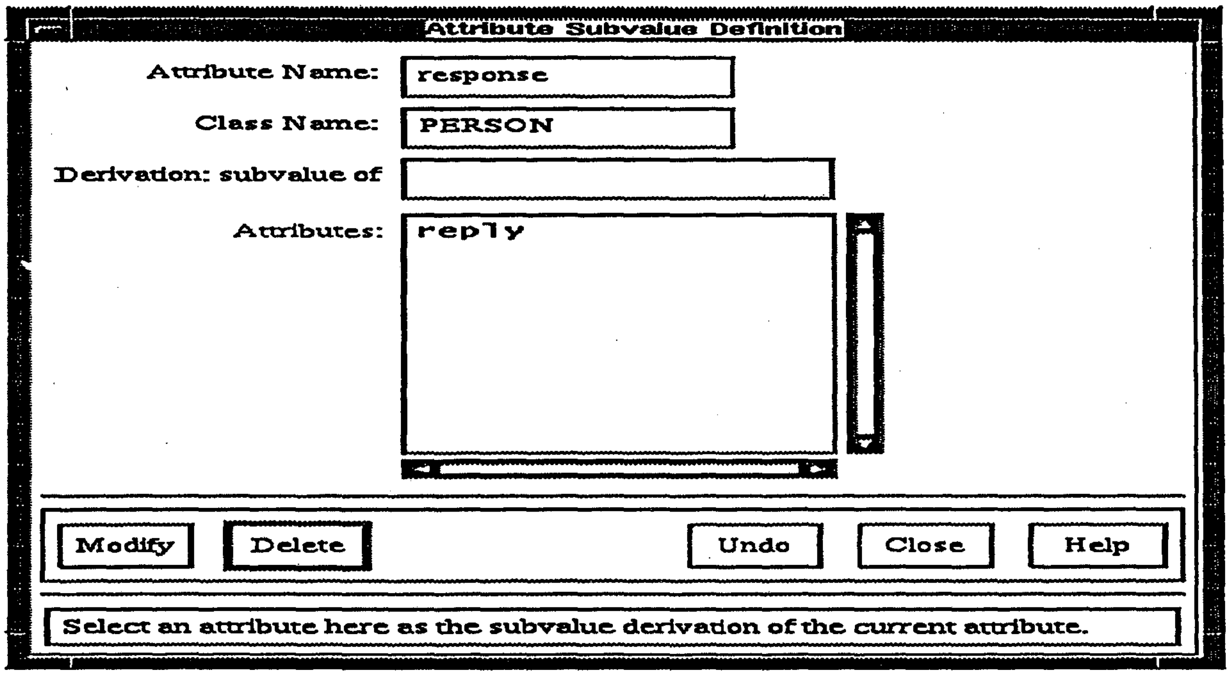
The **Attributes** Listbox displays all simple or component attributes of class PERSON that are non-derived or derived by inverse derivation and that are abstract. Click resume on the **Attributes** Listbox, and it will be displayed in the **Composition Derivation** area. The value classes of resume will be displayed in the **Value Classes** Listbox. For every value class selected in the **Value Classes** Listbox, its attributes that are non-derived or derived by inverse derivation will be displayed in the **Attributes** Listbox. If you click on an attribute in the **Attributes** Listbox, the same composition procedure is repeated. Click on candidate, and then click on reply. The **Attributes** Listbox now is empty because there is no valid attribute associated with class LETTER.

Click **New** in the **Attribute Composition Definition**, and then click **Modify** in the **Simple Attribute Definition** windows in order to save the changes.

3.5.6 Specifying Attribute Subvalue Derivations

First specify an object class MEMORANDUM with superclass LETTER, and then create a simple attribute response for the object class PERSON, and associate this attribute with the abstract value class MEMORANDUM. Click the **New** button on the **Simple Attribute Definition** window and select **subvalue** in the **Define Derivation** option menu. The **Attribute Subvalue Definition** window is brought up.

All simple or component attributes of class PERSON that can be defined as subvalues of attribute response are displayed in the **Attributes** Listbox. For the current schema, only attribute reply is dis-

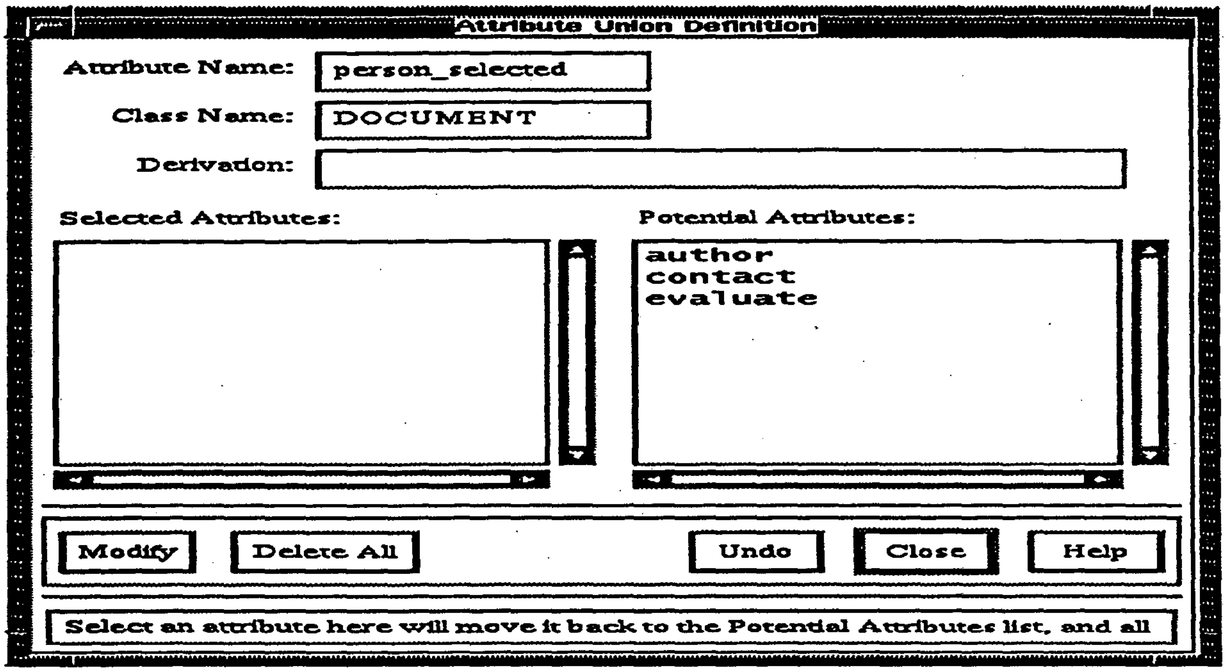


Select an attribute here as the subvalue derivation of the current attribute.

played in the listbox. The value class of response (i.e., MEMORANDUM) is an immediate subclass of the value class of attribute reply (i.e., LETTER). Click reply on the **Attributes** Listbox, and it will be displayed in the **Derivation: subvalue of** area. Click both **Modify** in the **Attribute Subvalue Definition** and the **Simple Attribute Definition** windows in order to save the changes.

3.5.7 Specifying Attribute Union Derivations

Specify three simple attributes for object class DOCUMENT: author, with abstract value class PERSON, evaluate, with abstract value class EVALUATE, and contact, with abstract value class TELEPHONE.



Attribute Name: person_selected

Class Name: DOCUMENT

Derivation:

Selected Attributes:

Potential Attributes:
author
contact
evaluate

Select an attribute here will move it back to the Potential Attributes list, and all

Then specify another simple attribute, person_selected, and associate this attribute with abstract value class PERSON or EVALUATE or TELEPHONE. Click the **New** button in the **Simple Attribute Definition** window and select **union** in the **Define Derivation** option menu. The **Attribute Union Definition** window is brought up.

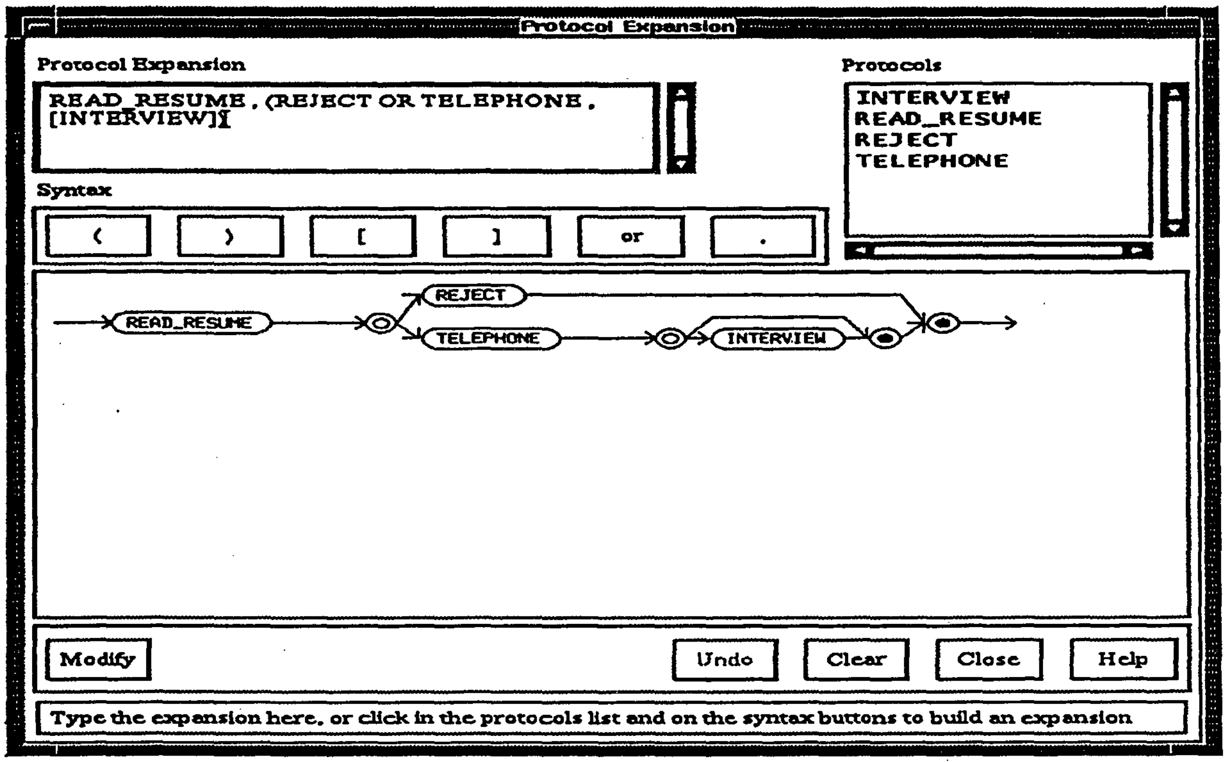
Attributes author, contact, and evaluate will be displayed in the **Potential Attributes** Listbox. The value class for attribute person_selected is the union of the value classes of attributes author, contact, and evaluate. Click on each of the attributes displayed in the **Potential Attributes** Listbox, and they will be entered into the **Derivation** area and listed in the **Selected Attributes** Listbox. Click **Modify** in the **Attribute Union Definition** and in the **Simple Attribute Definition** windows in order to save the changes.

3.6 Specifying Protocol Expansions

Protocol EVALUATE can be expressed in terms of (expanded into) simpler (sub-)protocols (steps) that are involved in the evaluation process.

First, specify an additional protocol class called REJECT representing the writing of rejection letters. After specifying REJECT, open the EVALUATE **Protocol Class Definition** window. Click the **Define Expansion** button in order to bring up the **Protocol Expansion** window.

The **Protocol Expansion** area is empty, and the **Protocols** Listbox lists four protocols that can potentially be involved in this expansion. Note that EVALUATE is not listed because a protocol cannot be defined in terms of itself. The expansion can be specified by typing directly in the **Protocol Expansion** area or by using the **Syntax** buttons and selecting protocols from the **Protocols** Listbox.



READ_RESUME, (REJECT OR TELEPHONE, [INTERVIEW])

INTERVIEW
READ_RESUME
REJECT
TELEPHONE

Type the expansion here, or click in the protocols list and on the syntax buttons to build an expansion

The evaluation process consists of reading a resume, then either rejecting the applicant immediately or deciding to telephone for arranging an interview. If the telephone conversation is not successful, you may not wish to arrange an interview. Thus, EVALUATE is expanded as follows:

READ_RESUME, (REJECT OR TELEPHONE, (INTERVIEW))

The parentheses group elements together. Commas represent sequential steps: first read the resume, then either reject or telephone. The square parentheses indicate an optional step: an interview may or may not be arranged. This expansion can be expressed as follows using the **Syntax** buttons: click on

READ_RESUME in the **Protocols** Listbox, click the comma button, click the left parenthesis button, click on REJECT, click the **or** button, click on TELEPHONE, click the comma button, click the left square parenthesis button, click on INTERVIEW, click the right square parenthesis button, click the right parenthesis button.

Click on **Modify** to associate EVALUATE with the new protocol expansion. The new protocol expansion will appear in the **Protocol Expansion** area in EVALUATE **Protocol Class Definition** window. Moreover, a graphical representation of the expansion is represented graphically in the drawing area of the **Protocol Expansion** window. Remember to click **Modify** in the **Protocol Class Definition** window to save the change.

Troubleshooting: Without parentheses the expansion (READ_RESUME, REJECT OR TELEPHONE, (INTERVIEW)) is quite different: it specifies the evaluation process as consisting of the following sequence of steps: either read the applicant's resume **and** then reject the applicant, or telephone the applicant and possibly arrange an interview (operator "," has higher precedence over **or**)!

Now try reverting all parentheses into square parentheses in the expansion text. Click **Modify.** The protocol expansion parser cannot interpret this new expression, and warns you of the error by indicating the location of the error.

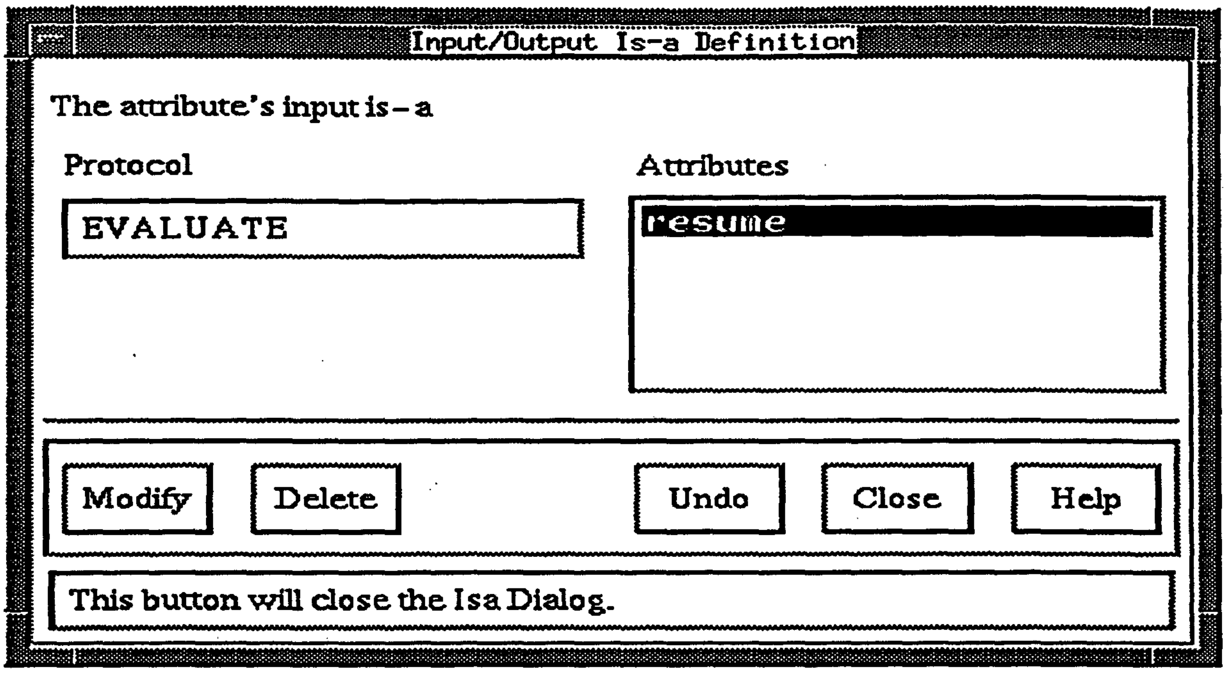
3.7 Specifying Protocol Connections

Connections between protocols are specified using input and output attributes. If an input or output attribute ***A*** of a protocol class ***P*** is identical to an input or output attribute ***B*** of one of the subprotocols of ***P***, ***Q*** (i.e., ***Q*** is involved in the expansion of ***P***, and ***B*** represents ***A*** in ***Q***), then ***B*** is specified as an **Input is-a** or **Output is-a** relative to ***A***. If an input attribute ***A*** of protocol class ***P*** takes its input from an output attribute ***B*** of another protocol class ***Q***, then ***A*** is specified as an **Input from** relative to ***B***.

3.7.1 Specifying Input or Output Is-a Connections

The EVALUATE protocol class should have an input attribute representing resumes. Bring up the EVALUATE **Protocol Class Definition** window, and specify an input attribute called resume with abstract value class RESUME.

Clearly, input attribute resume of sub-protocol READ_RESUME is identical to input attribute resume of the higher-level protocol EVALUATE. Bring up the READ_RESUME **Protocol Class Definition** window, and then the **Input/Output Attribute Definition** window for resume. Select **Input is-a** from the **Define Connection** option menu. The **Input/Output Is-a Definition** window is brought up.



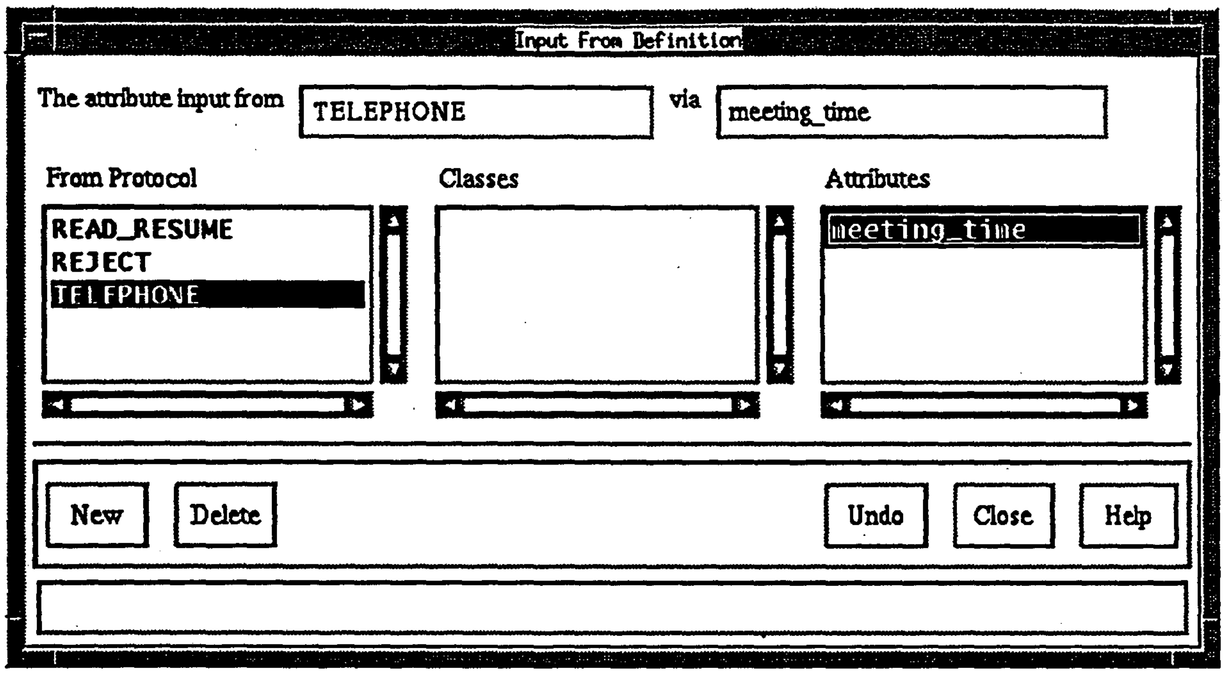
This button will close the Isa Dialog.

The higher-level protocol class, in this case EVALUATE, is automatically displayed in the **Protocol** area. The input attributes of this higher-level protocol class (in this case only one) are listed in the **Attributes** Listbox. Select resume from this listbox, click **Modify**, and close this window. The input is-a definition is displayed in the **Connection** area of the **Input/Output Attribute Definition** window. Remember to click **Modify** before closing **Input/Output Attribute Definition** window.

An input attribute can have an input is-a connection if it is part of a higher level protocol, and the connection must refer to an input attribute of that higher level protocol. In the example above the names and value classes are identical for the connected attributes, but this is not a requirement. Similar constraints apply to the output is-a connection. As an exercise, create an output attribute decision for the EVALUATE protocol class with a controlled value class DECISION, and connect output attribute decision of protocol class INTERVIEW to it.

3.7.2 Specifying Input From Connections

The TELEPHONE protocol class has a meeting_time output attribute. A similar attribute can be considered as an input attribute for the INTERVIEW protocol class. Accordingly, specify an input attribute time for protocol class INTERVIEW, and associate it with primitive value class DATETIME. In the **Input/ Output Attribute Definition** window select **Input from** from the **Define Connection** option menu. The **Input From Definition** window is brought up.


Input From Definition

The attribute input from TELEPHONE via meeting_time

From Protocol: READ_RESUME, REJECT, TELEPHONE

Classes

Attributes: meeting_time

The **From Protocol** Listbox in this window lists all the protocol classes having output attributes that can be connected to the current attribute. Select TELEPHONE in the **From Protocol** Listbox. The name of the selected protocol appears in the **The attribute input from** area, and the **Attributes** Listbox lists the output attributes of the selected protocol class: in this case, meeting_time is listed. Select this attribute by clicking on it; its name is displayed in tne **via** area. Click **New**, and the input from definition is displayed in the **Connection** area of the **Input/Output Attribute Definition** window. Remember to click **Modify** before closing the **Input/Output Attribute Definition** window.

An input attribute of protocol class ***P*** can have an **Input from** connection either (1) if ***P*** is not a sub-protocol of a higher-level protocol class, or (2) if ***P*** is a sub-protocol of protocol class ***Q***, and there are other sub-protocols of ***Q*** preceding ***P*** in the expansion definition. The **via** attribute must be an output attribute of the **From Protocol.** In some cases, there is a sequence of via attributes. In order to continue adding via attributes, the most recent via attribute must be of abstract value class and the new via attribute must be an attribute of that class.

4 OPM Schema Editor Windows

This section describes every window one may encounter while using the editor. The layout of every window, buttons in the window, and the functionality of each button are described.

We start by presenting the general window structure, which is generic for all the windows in the editor (see Figure 1). Then we present the OPM Schema editor window flow diagram in Figure 2.The main menu, the functionality of each menu item, and how to use the main menu to start or end an application is then explained. Finally, we will concentrate on specific functions for every individual window.

4.1 General Window Structure

All the windows in the OPM Editor follow the general window structure shown in Figure 1.

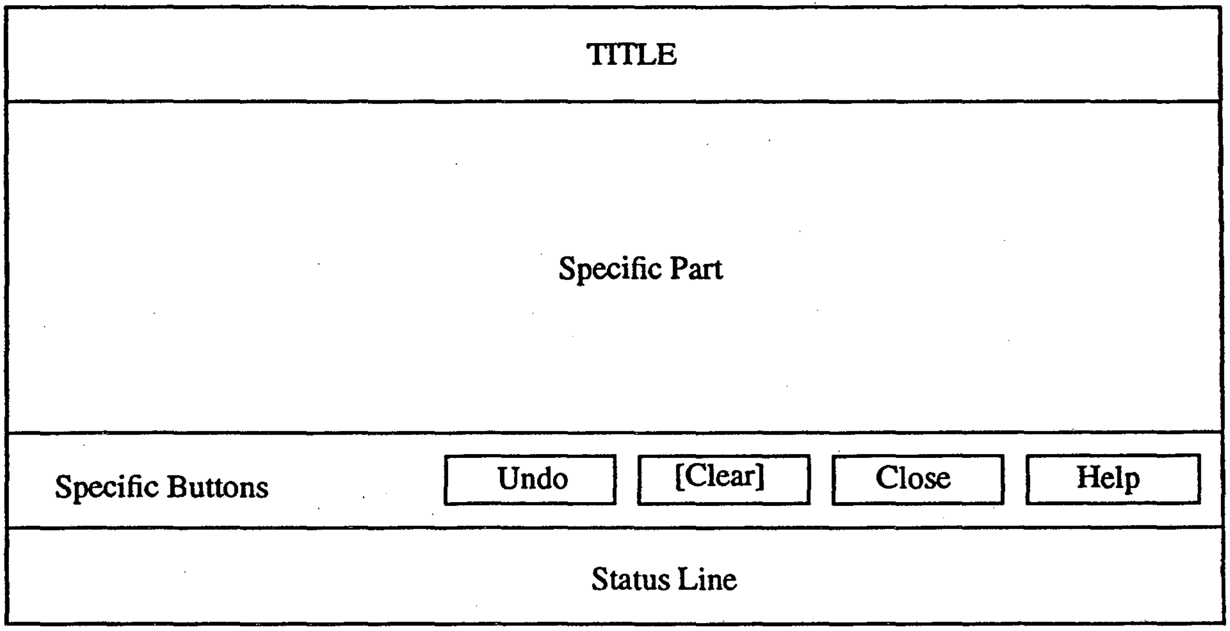


FIGURE 1. **General Window Structure**

A window can be divided into two parts: a window-specific part and a generic part. The latter is (almost) identical in all the windows except the main window. In the generic part of a window, there are:

1. a window title
2. a status line with instructions for next action;
3. an **Undo** button (undo the previous action);
4. a **Close** button (close this window);

5. a **Help** button (provide help for this window); and
6. an optional **Clear** button (clear this window).

All the command buttons are shown on the same row. Buttons common to all the windows are grouped to the right hand side of the command button row.

4.2 Window Flow

The window flow for the OPM Editor is shown in Figure 2. Each box represents an editor window. Two boxes are connected by an arrow if the window represented by the box to which the arrow is directed, can be reached from the window represented by the box where the arrow starts, by selecting a menu item or clicking a button.

4.3 Main Window

The main window of the **OPM Schema Editor** is shown in Figure 3.

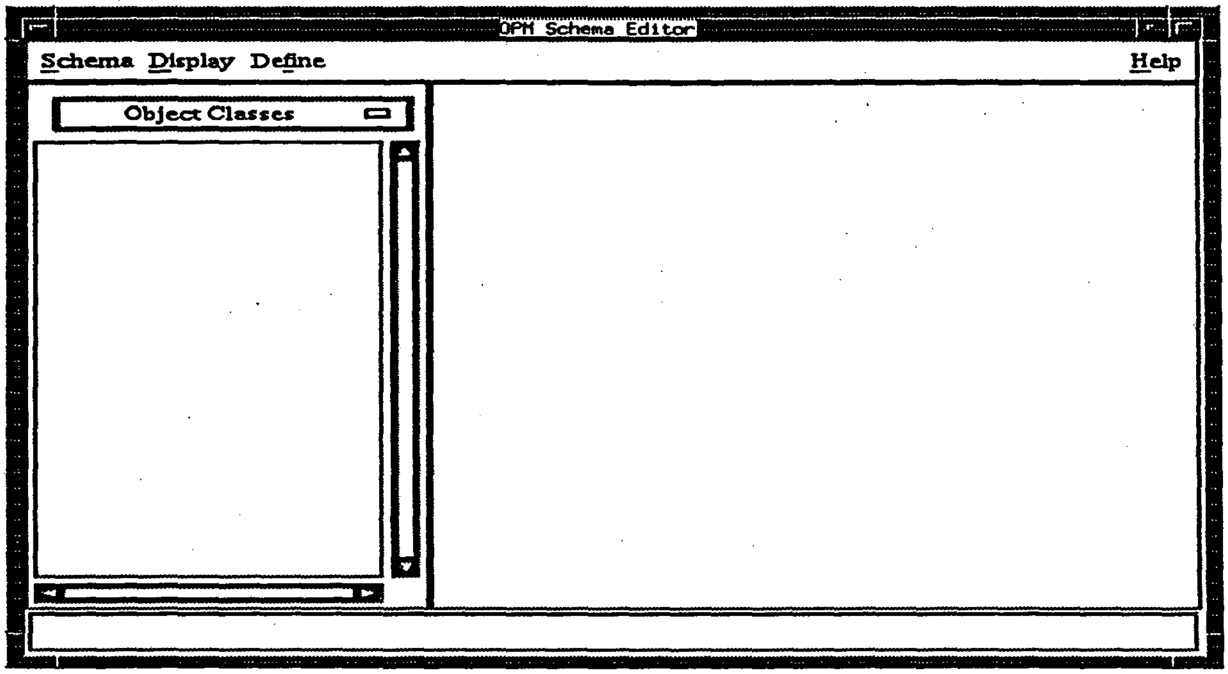


FIGURE 3. **OPM Schema Editor** Main Window

We list only the functions of all the menu items in this subsection.

1. **Schema**
   (a) **New** - create a new OPM schema.

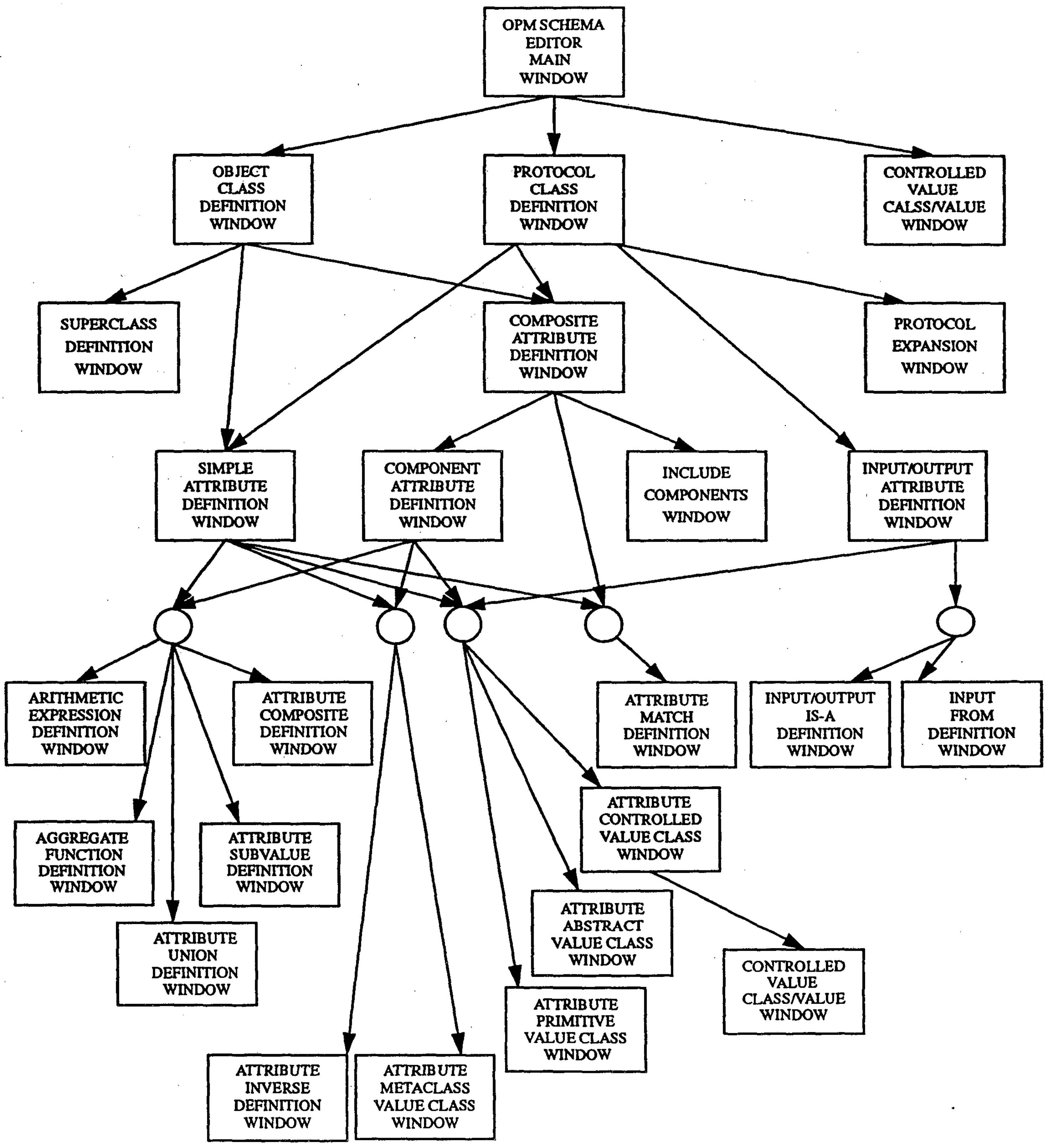

OPM SCHEMA EDITOR MAIN WINDOW
OBJECT CLASS DEFINITION WINDOW
PROTOCOL CLASS DEFINITION WINDOW
CONTROLLED VALUE CALSS/VALUE WINDOW
SUPERCLASS DEFINITION WINDOW
COMPOSITE ATTRIBUTE DEFINITION WINDOW
PROTOCOL EXPANSION WINDOW
SIMPLE ATTRIBUTE DEFINITION WINDOW
COMPONENT ATTRIBUTE DEFINITION WINDOW
INCLUDE COMPONENTS WINDOW
INPUT/OUTPUT ATTRIBUTE DEFINITION WINDOW
ARITHMETIC EXPRESSION DEFINITION WINDOW
ATTRIBUTE COMPOSITE DEFINITION WINDOW
ATTRIBUTE MATCH DEFINITION WINDOW
INPUT/OUTPUT IS-A DEFINITION WINDOW
INPUT FROM DEFINITION WINDOW
ATTRIBUTE CONTROLLED VALUE CLASS WINDOW
AGGREGATE FUNCTION DEFINITION WINDOW
ATTRIBUTE SUBVALUE DEFINITION WINDOW
ATTRIBUTE ABSTRACT VALUE CLASS WINDOW
ATTRIBUTE UNION DEFINITION WINDOW
CONTROLLED VALUE CLASS/VALUE WINDOW
ATTRIBUTE PRIMITIVE VALUE CLASS WINDOW
ATTRIBUTE INVERSE DEFINITION WINDOW
ATTRIBUTE METACLASS VALUE CLASS WINDOW

FIGURE 2. **Window Flow** Diagram

   (b) **Open** - open an existing schema.
   (c) **Append** - append an existing schema to the current one.
   (d) **Save** - save current schema.
   (e) **Save As** - save current schema to a file.
   (f) **Print OPM in Latex**- output current OPM schema into a Latex file.
   (g) **Print in PostScript** - output current OPM schema into a PostScript file.
   (h) **Quit** - quit OPM editor.

2. **Display**
   (a) **Clear** - reset (clear) the display window.
   (b) **Class Hierarchy** - the object class hierarchy of the current schema is graphically displayed if this mode is on.
   (c) **Class Links** - classes and their attributes are graphically displayed if this mode is on.
   (d) **Detailed Links** - attributes of subprotocols involved in an expansion are also graphically displayed if this mode is on.
3. **Define**
   (a) **OPM Object Class** - define a new object class. After this option is selected, a new (blank) **Object Class Definition** window (see Figure 4) is brought up for adding a new object class. If an existing object class is selected by double clicking the class name in the **Object Classes** Listbox, then the definition of the selected object class will be displayed in the **Object Class Definition** window (Figure 3).
   (b) **OPM Protocol Class** - define a new protocol class (similar operation as define object class, but using **Protocol Class Definition** window (see Figure 5) and **Protocol Classes** Listbox).
   (c) **Controlled Value Class** - define a new controlled value class (similar operation as define object class, but using **Controlled Value Class/Value** window (see Figure 13) and **Controlled Value Classes** Listbox).
4. **Help**
   (a) **About.**
   (b) **Help.**

The main window contains a listbox for listing in alphabetical order object classes, protocol classes and controlled value classes. The listbox is headed by an option menu that allows selecting one of the following display types:

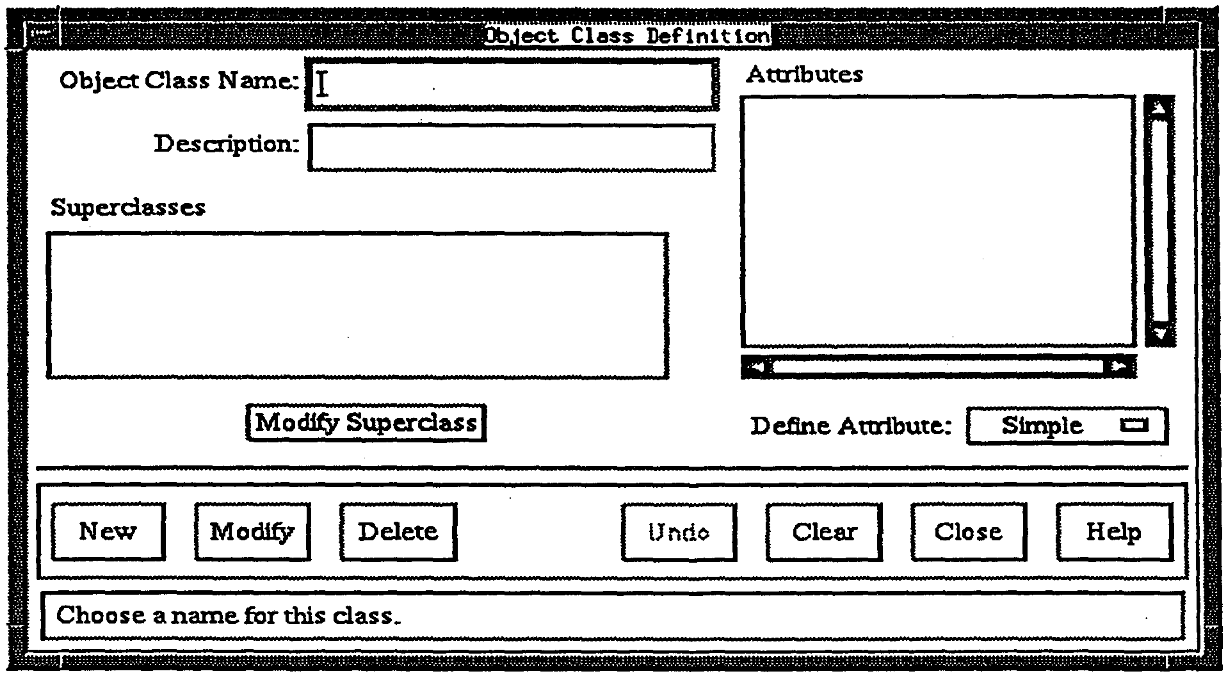
1. **Object Classes** for displaying the object class names.
2. **Protocol Classes** for displaying the protocol class names.
3. **Controlled Value Classes** for displaying the controlled value class names.

If an object class is selected in the listbox (and the **Class Links** mode is on), a diagram representing the selected class, its superclasses, its subclasses, and its attributes is displayed in the main window drawing area. If a protocol class is selected in the listbox, a graphical representation of its expansion (if any) is displayed in the main window drawing area. These diagrammatic representations have buttons representing class names. Double clicking on these buttons, just like double clicking on list elements, brings up the class definition window for the selected class.

If a controlled value class is selected in the listbox (and the **Class Links** mode is on), then the controlled value class name together with all the values and/or ranges of this class are displayed in the main window drawing area.

4.4 Define Object Class

An object class has a class name, an optional class description, and a set of associated attributes. A specialization class (subclass) has one or more superclasses.



Choose a name for this class.

FIGURE 4. **Object Class Definition** Window

A new object class can be added and an existing object class can be modified or deleted using the **Object Class Definition** window (Figure 4)

4.4.1 Add Object Class

Before being added, an object class must be associated with a non-null distinct (unique) class name. The class name must be in upper case letters and cannot exceed 32 characters. Class description is optional and cannot exceed 256 characters.

The attributes and/or superclasses of an object class can be specified either before adding the class, or after adding the class using the **New** button.

An object class can have multiple object superclasses. All the superclasses defined for the current object class are listed in the **Superclasses** Listbox. After **Modify Superclass** button is used, a **Superclass Definition** window (Figure 6) is brought up for adding new superclasses or for deleting existing superclasses.

The attributes of the current object class are listed in the **Attributes** Listbox. An attribute can be added/modified/deleted using the **Define Attribute** button. If no attribute in the listbox is highlighted, then using the **Simple** button brings up a new **Simple Attribute Definition** window (Figure 8), and using the **Composite** button brings up a new **Composite Attribute Definition** window (Figure 9). If there is a selected attribute, then using the above mentioned buttons brings up an attribute window for the selected attribute.

4.4.2 Modify Object Class

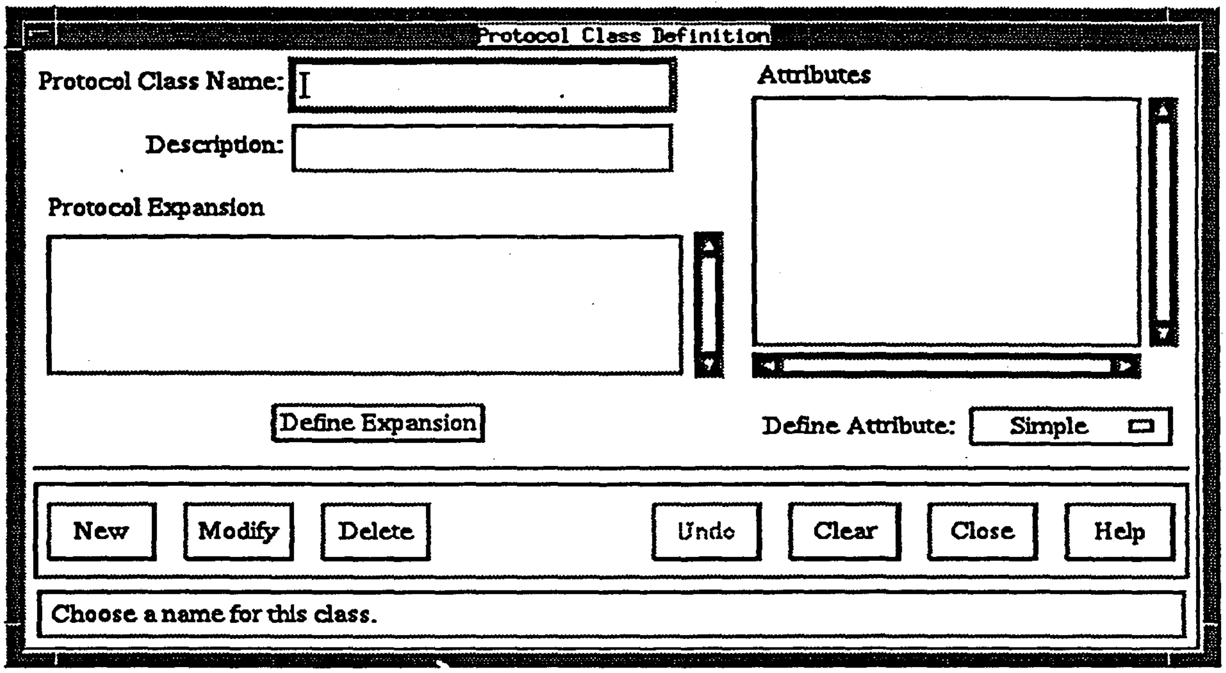
Object class names, descriptions, superclasses and attributes can all be modified. The text in **Object Class Name** or **Description** area can be edited in order to change the class name or description, respectively. Clicking **Modify Superclass** or **Define Attribute** button allows modifying the superclasses or attributes of a class. The procedure to modify the superclasses and attributes is the same as the procedure described in previous subsection. After all the desired changes have been made, the schema is updated using the **Modify** button.

4.4.3 Delete Object Class

Delete button allows deleting the current object class definition after the user confirms the action.

4.5 Define Protocol Class

A protocol class has a class name, an optional class description, and a set of associated attributes. If the protocol can be expanded into several subprotocols, then a protocol expansion is also specified. A new protocol class can be added and an existing protocol class can be modified or deleted using the **Protocol Class Definition** window (Figure 5).



Choose a name for this class.

FIGURE 5. **Protocol Class Definition** Window

4.5.1 Add Protocol Class

Before being added, a protocol class must be associated with a distinct non-null class name. The class name must be in upper case letters and cannot exceed 32 characters. Class description is optional and cannot exceed 256 characters.

The attributes of a protocol class can be specified before adding the class, or after the class has been added using the **New** button.

The attributes of the current protocol class are listed in the **Attributes** Listbox. An attribute can be added/modified/deleted using the **Define Attribute** button. If no attribute in the listbox is highlighted, then using the **Simple** button brings up a new **Simple Attribute Definition** window (Figure 8), using the **Composite** button brings up a new **Composite Attribute Definition** window (Figure 9), and using the **Input/Output** button brings up a new **Input/Output Attribute Definition** window (Figure 12). If there is a selected attribute, then using the above mentioned buttons brings up an attribute window for the selected attribute.

Protocol expansion can be specified only after an expanded protocol has been added to the schema using the **New** button. The protocol expansion is displayed in the **Protocol Expansion** area. Click the **Define Expansion** button to bring up a **Protocol Expansion** window (Figure 7) for specifying or modifying a protocol expansion.

4.5.2 Modify Protocol Class

Protocol class names, descriptions, expansions and attributes can all be modified. In order to change a class name or a description, the text can be edited in the **Protocol Class Name** or the **Description** area, respectively. Click **Define Expansion** or **Define Attribute** button to modify protocol expansion or attributes of the class. The procedure to modify expansion and attributes is the same as the procedure described in previous subsection. Changes are finalized (i.e., recorded as schema updates) using the **Modify** button.

4.5.3 Delete Protocol Class

The current protocol class can be deleted using **Delete** button, after the user confirms the action.

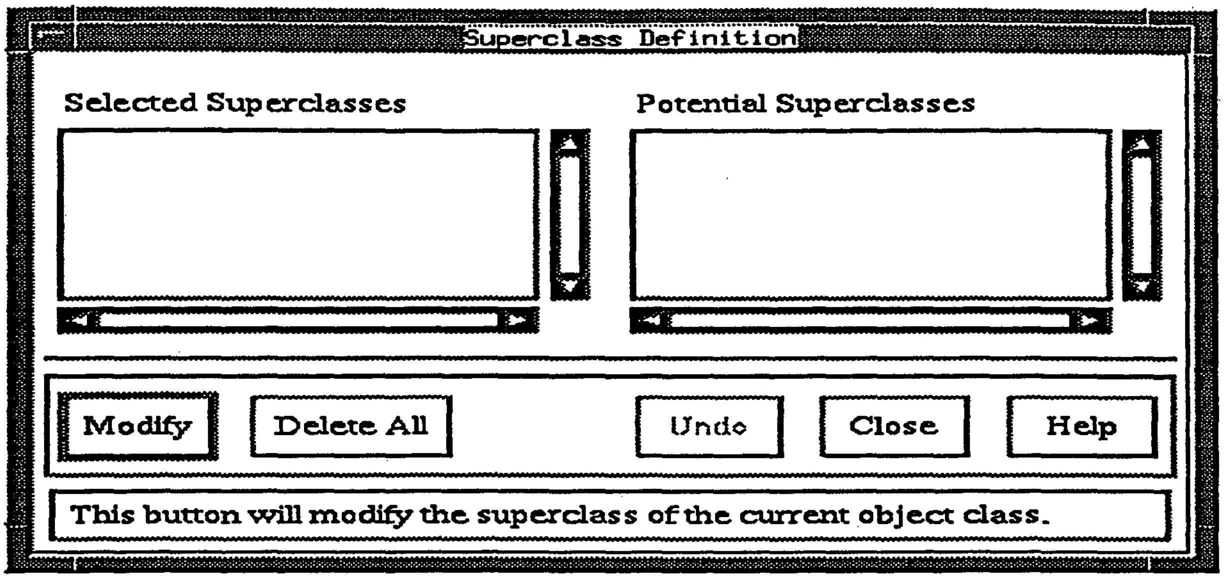
4.6 Define Superclass

A subclass has one or more superclasses, and it inherits all the attributes of its superclasses.A class cannot be specified as a superclass of itself. Moreover, subclasses of a class cannot be specified as superclasses of this class.

The **Superclass Definition** window (Figure 6) is used to define superclasses of an object class. There are two lists of class names displayed in this window. The **Selected Superclasses** Listbox and the **Potential Superclasses** Listbox are complementary. An object class (except for the current one and its subclasses) is listed in exactly one of the listboxes. The **Selected Superclasses** Listbox contains all the superclasses of the current class. Transitive superclasses can be either included or not. Clicking on a class name in the **Selected Superclasses** Listbox moves this class from the **Selected Superclasses** Listbox to the **Potential Superclasses** Listbox, and vice versa.

The **Delete All** button allows clearing the **Selected Superclasses** Listbox, and moving all the object classes to the **Potential Superclasses** Listbox.

After all the superclasses are properly selected, the **Modify** button must be used in order to update



This button will modify the superclass of the current object class.

FIGURE 6. **Superclass Definition** Window

the **Object Class Definition** window with the new superclass information.

4.7 Define Protocol Expansion

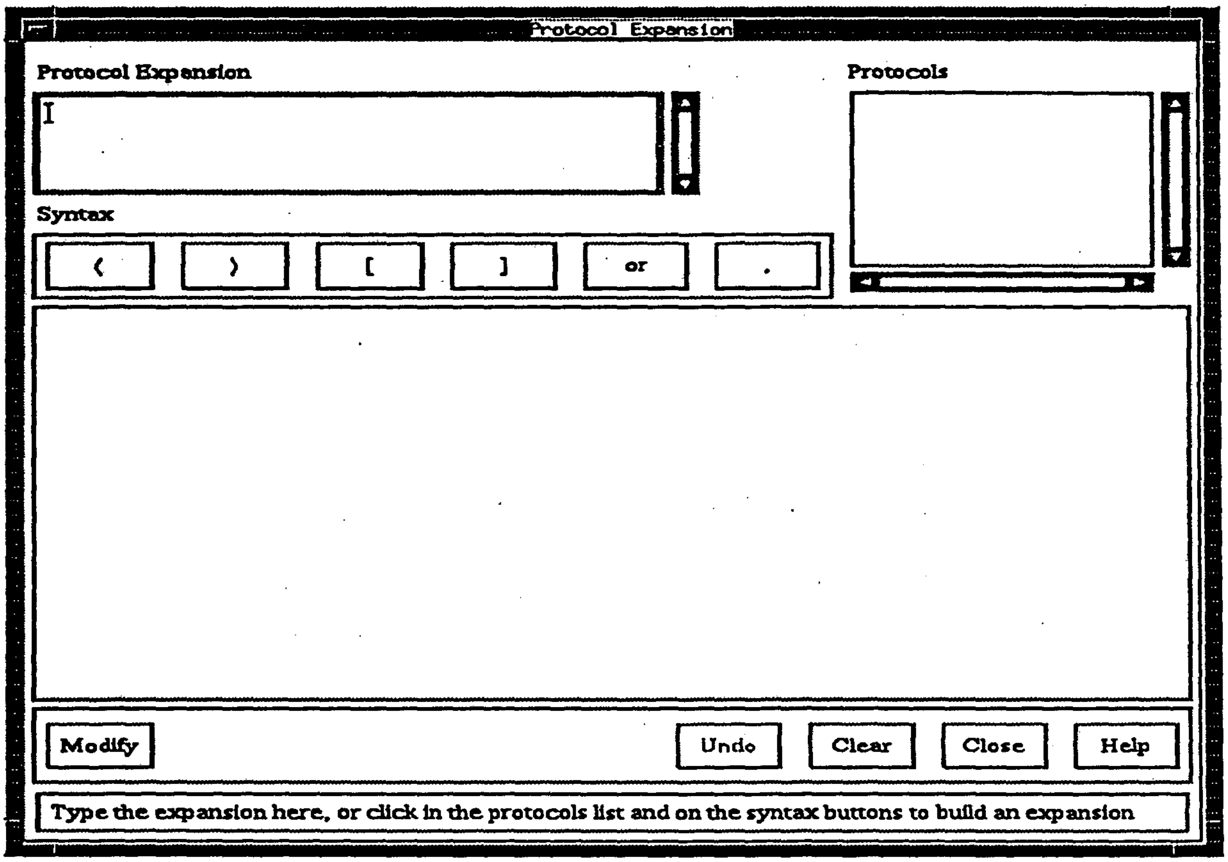
Protocol expansion allows specifying *alternative protocols*, *sequences of protocols*, and *optional protocols*; "*or*", ",", "[" and "]" are used to denote alternative, sequences of, and optional protocols, respectively, and parentheses are used for specifying complex protocol compositions. Operator "," has higher precedence than *or*. For example, if P is a protocol whose expansion is $(A, B, [C])$ *or* D, then protocol P is defined as either (i) the sequence of protocols A followed by B and followed by optional protocol C, or (alternative) (ii) protocol D alone. The protocol expansion must be acyclic, that is, if a protocol class P_i is involved in the expansion of protocol class P_j, then P_j cannot be involved in the expansion of P_i or any subprotocol of P_i.

The protocol expansion is displayed in the **Protocol Expansion** working area in the **Protocol Expansion** window (Figure 7). Protocol expansion can be directly specified in the working area, or can be specified using the listbox and function buttons provided for defining protocol expansion.

The protocol class names that can appear in the protocol expansion of the current protocol class P_i are listed in the **Protocols** Listbox. A protocol P_j can be used in the expansion of P_i if P_i and P_j are different, and P_i does not appear in the expansion of P_j or any subprotocol of P_j (transitively).

When a protocol name is clicked in the listbox, the selected protocol class name will be highlighted and inserted into the **Protocol Expansion** working area. Clicking the six function buttons ("*or*", ",",

"[", "]", "(", ")") causes the corresponding symbol to appear in the working area. In a protocol expansion expression, two protocol names must be separated by operators "*or*" or ",". If a protocol name is appended immediately after another protocol name in the expansion, then the editor automatically adds a "," between the two protocol names.



Type the expansion here, or click in the protocols list and on the syntax buttons to build an expansion

FIGURE 7. **Protocol Expansion** Window

A protocol can have at most one higher level protocol. That is, if a protocol class P_j is contained in the expansion of protocol class P_i, then P_j cannot be contained in the expansion of other protocol classes.

A graphical representation of the protocol expansion is also displayed in the "protocol drawing area". This drawing area will be updated every time a new and correct protocol expansion is entered after **Modify** button is pressed.

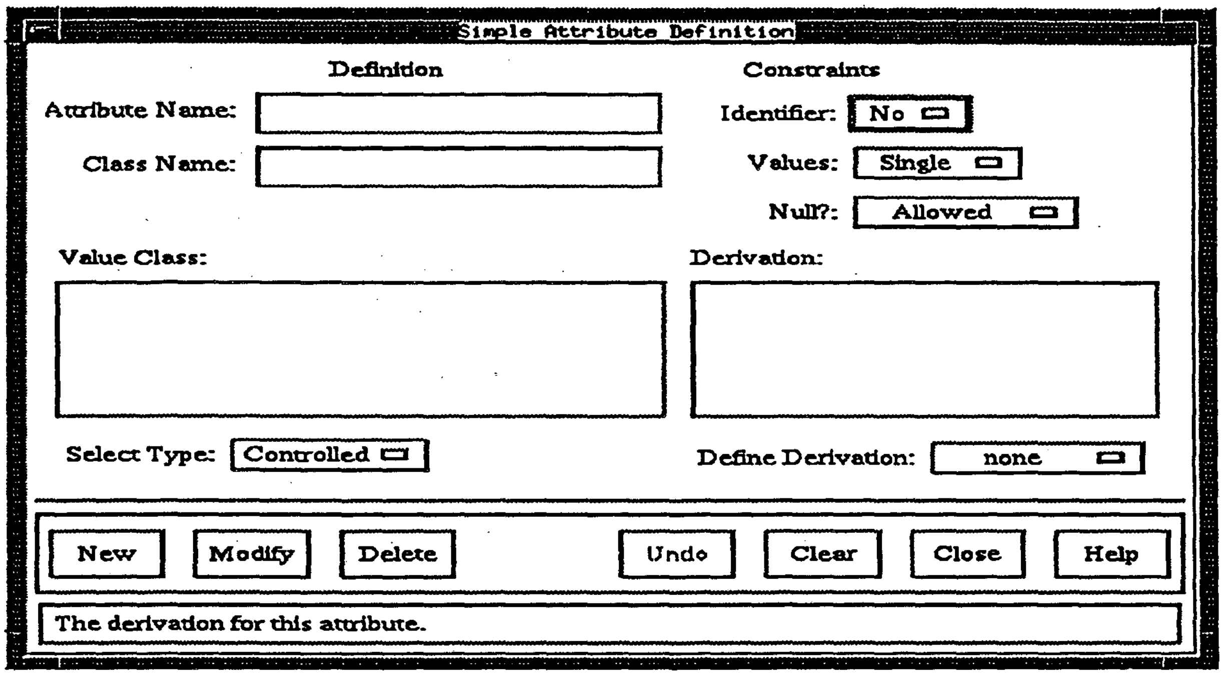
Clicking the **Clear** button clears the **Protocol Expansion** working area and the "protocol drawing area".

After the protocol expansion is specified, the **Modify** button must be used in order to update the

Protocol Class Definition window with the new protocol expansion.

4.8 Define Simple Attribute

Each simple attribute has an attribute name, an associated value class, and a set of attribute constraints. If the attribute is derived, then an attribute derivation is also specified.



The derivation for this attribute.

FIGURE 8. **Simple Attribute Definition** Window

4.8.1 Add Simple Attribute

The name of the current (object or protocol) class is displayed in the **Class Name** area. An attribute name, a value class and associated constraints must be provided before a new attribute can be added.

All the explicitly defined and inherited attributes of a class must have distinct, non-null names. Attribute names are in lower case letters and cannot exceed 32 characters.

The **Yes** option for the **Identifier** button indicates that the attribute is (part of) an object identifier; **No** indicates that the attribute is not (part of) an object identifier. The **Single** option of the **Values** button indicates that the attribute can have only one value associated with it; if an attribute is associated with a set of values, then the **Multiple** option must be selected.

The **Allowed** option of the **Null** button indicates that the attribute can have null values; if the attribute is not allowed to have null values, then the **Not Allowed** option must be selected. Note that an attribute that is (part of) an object identifier, cannot have null values.

Each attribute must have an associated value class. The value class of the current attribute is listed in the **Value Class** Listbox. The **Select Type** option menu can be used for defining or modifying the value class. An attribute can be associated with one of the following four types of value classes:

1. **Controlled: An Attribute Controlled Value Class** window (Figure 14) is brought up.
2. **Primitive: An Attribute Primitive Value Class** window (Figure 15) is brought up.
3. **Abstract**: An **Attribute Abstract Value Class** window (Figure 16) is brought up.
4. **Metaclass: An Attribute Metaclass Value Class** window (Figure 17) is brought up.

When the type of value class is changed, the user is informed that the previously defined value class is destroyed. After the attribute name, value class and attribute constraints are specified, the attribute is associated with the current class using the **New** button.

For a derived attribute the derivation rule can be specified after the attribute has been associated with a class using the **New** button.

The derivation rule of an attribute (if any) is displayed in the **Derivation** box. In order to define or modify a derivation rule, the **Define Derivation** option menu must be used. An attribute can have at most one derivation. There are eight options for attribute derivations:

1. **none**: This attribute has no derivation. A previously defined derivation will be removed.
2. **arithmetic**: An **Arithmetic Expression Definition** window (Figure 20) is brought up.
3. **aggregate**: An **Aggregate Function Definition** window (Figure 21) is brought up.
4. **composition**: An **Attribute Composition Definition** window (Figure 22) is brought up.
5. **inverse**: An **Attribute Inverse Definition** window (Figure 18) is brought up.
6. **match**: An **Attribute Match Definition** window (Figure 19) is brought up.
7. **subvalue**: An **Attribute Subvalue Definition** window (Figure 23) is brought up.
8. **union**: An **Attribute Union Definition** window (Figure 24) is brought up.

If an attribute has been previously defined as derived and its derivation type is changed (e.g., from an inverse attribute to an arithmetic expression derived attribute), then a confirmation of the change will be required.

Note that an attribute which is (part of) an identifier cannot be a derived attribute. Moreover, only simple, single-valued attributes can have arithmetic expression or aggregate function derivation. These attributes must be primitive and associated with one of the following value classes: INTEGER, SMALLINT, REAL, FLOAT, or MONEY.

4.8.2 Modify Simple Attribute

Attribute names, value classes, constraints and derivation rules can be modified. Changes one finalized (i.e., are recorded as schema updates) using the **Modify** button.

The modification of attribute constraints, value classes, and derivation specifications are carried out as described in the subsection **Add a Simple Attribute** .

4.8.3 Delete Simple Attribute

An attribute can be deleted using the **Delete** button. Deletion is carried out only after the user confirms the action.

4.9 Define Composite Attribute

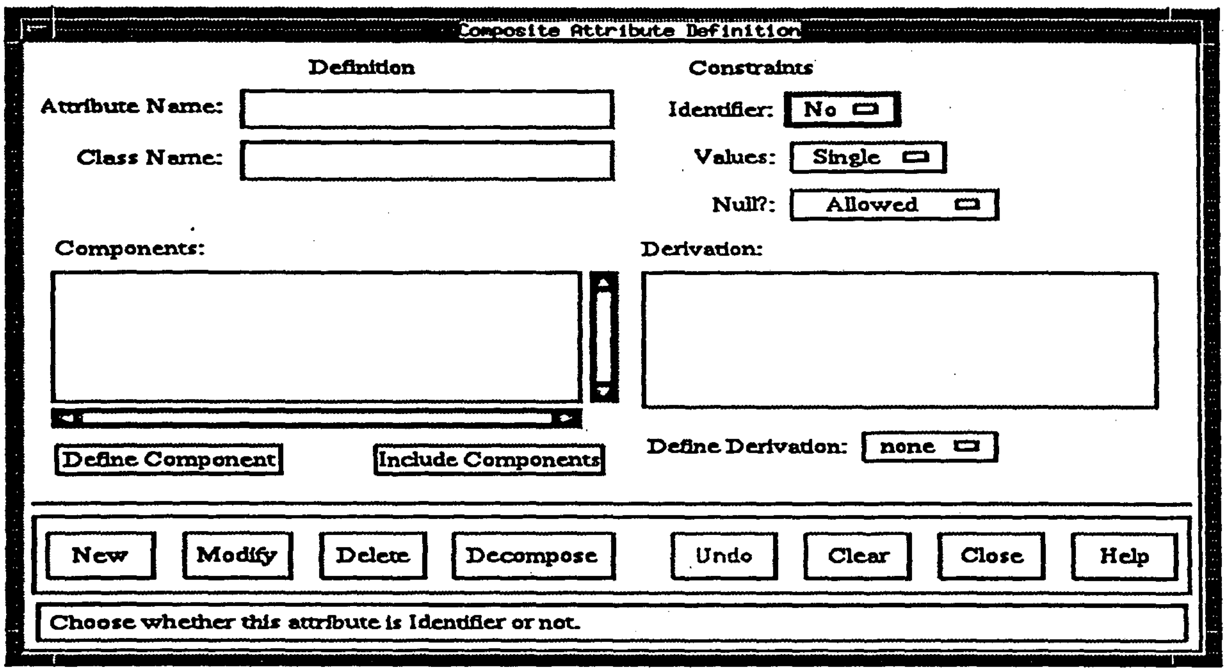
Each composite attribute has an optional attribute name, a set of component attributes, and associated constraints. The constraints on composite attribute are applied to each of the components. A composite attribute can be associated with an attribute matching derivation.

4.9.1 Add Composite Attribute

The name of current (object or protocol) class is displayed in the **Class Name** area. The name for a composite attribute is optional. If such a name is provided, then it must be unique within the class, in lower case letters, and must not exceed 32 characters. Constraints are defined as follows. The **Yes** option for the **Identifier** button indicates that the composite attribute is (part of) an object identifier; **No** indicates that the attribute is not (part of) an object identifier.

The **Single** option of the **Values** button indicates that every component attribute can have only one value associated with it; if all component attributes can be associated with sets of values, then the **Multiple** option must be selected. The **Allowed** option of the **Null** button indicates that the attribute can have null values. Note that if the attribute is (part of) an object identifier, then it cannot have null values.

The component attributes of a composite attribute are listed in the **Components** Listbox. Component attributes can be added/modified/deleted using the **Define Component** button. If a component attribute is highlighted, then **Define Component** button brings up a **Component Attribute Definition** window (Figure 10) for modifying the attribute. Otherwise, a new **Component Attribute Definition** window (Figure 10) is displayed for the definition of a new attribute. After all the necessary information (except the derivation) is specified, the attribute can be associated with the current class using the **New** button. If the composite attribute is derived, then the derivation can be specified after associating the attribute with the current class.



Choose whether this attribute is Identifier or not.

FIGURE 9. **Composite Attribute Definition** Window

The derivation rule of a composite attribute (if any) is displayed in the **Derivation** box. The derivation rule can be defined or modified using the **Define Derivation** button:

1. **none:** This attribute has no match derivation. A previously defined match derivation will be removed.
2. **match:** An **Attribute Match Definition** window (Figure 19) is brought up.

Whenever the derivation type for an attribute is changed, the user is asked to confirm the change. Note that an attribute which is (part of) an identifier cannot have a derivation.

Component attributes of a composite attribute can also be defined by including existing simple attributes of the target object or protocol class. Selecting **Include Components** button will bring up an **Include Components** window (Figure 11) for including simple attributes as components of the current composite attribute.

The **Decompose** button decomposes the composite attribute. That is, all the component attributes become simple attributes of the target class.

4.9.2 Modify Composite Attribute

Attribute names, component attributes, constraints and derivation rules can be modified. Changes are finalized (i.e., recorded as schema updates) using the **Modify** button.

The modification of attribute constraints, components, and derivation specification is carried out as described in the subsection **Add a Composite Attribute.**

4.9.3 Delete Composite Attribute

An attribute can be deleted using the **Delete** button. Deletion is carried out only after the user confirms the action.

4.10 Define Component Attribute

Each component attribute must have a distinct attribute name, and must be associated with a value class. A component attribute can have an attribute derivation.

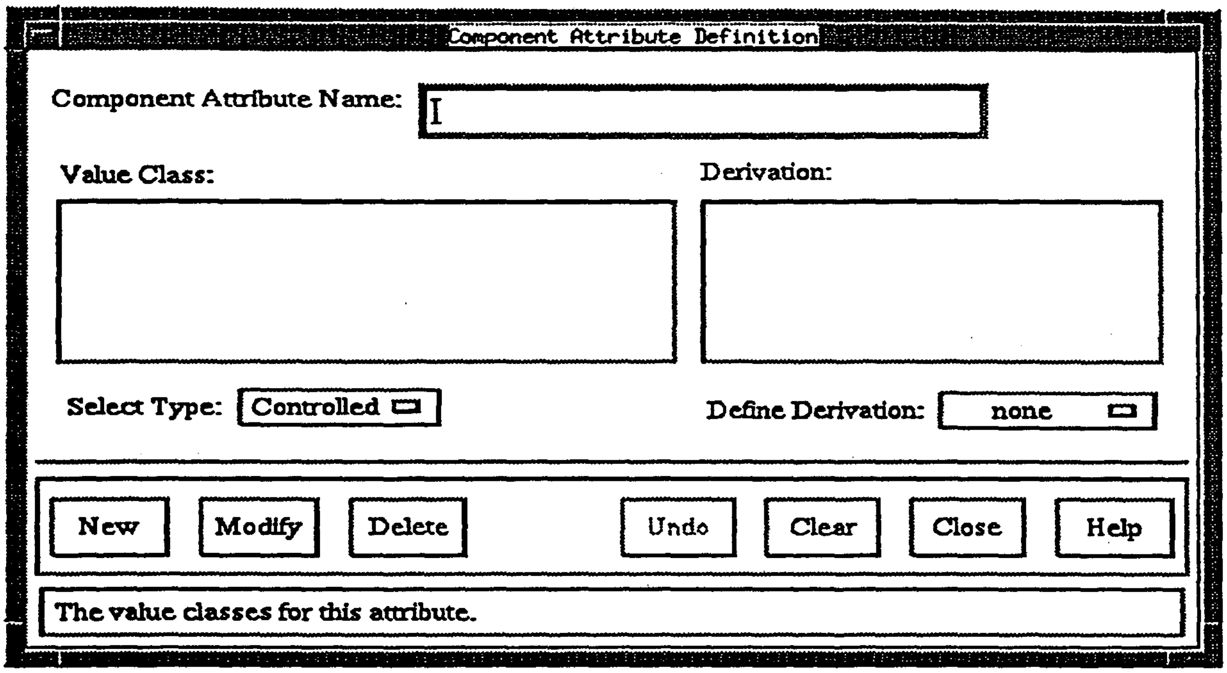
4.10.1 Add Component Attribute

In order to add a component attribute to the composite, a name for the component attribute must be specified first. Although the name of the composite attribute is optional, every component attribute of a composite attribute must have a name. Such a name must be distinct, in lower case letters, and should not exceed 32 characters.

Every component attribute must have an associated value class. The value class of the current component attribute is listed in the **Value Class** Listbox. The **Select Type** option menu can be used for defining or modifying the value class. A component attribute can be associated with one of the following four types of value classes:

1. **Controlled:** An **Attribute Controlled Value Class** window (Figure 14) is brought up.
2. **Primitive:** An **Attribute Primitive Value Class** window (Figure 15) is brought up.
3. **Abstract:** An **Attribute Abstract Value Class** window (Figure 16) is brought up.
4. **Metaclass:** An **Attribute Metaclass Value Class** window (Figure 17) is brought up.

When the type of value class is changed, the user is informed that the previously defined value class is destroyed.



The value classes for this attribute.

FIGURE 10. **Component Attribute Definition** Window

After all the necessary information (except the derivation) is specified, the component attribute is added to its composite attribute using the **New** button. If a component attribute has an attribute derivation, then this derivation can be specified after adding the attribute.

The derivation of a component attribute (if any) is displayed in the **Derivation** box. The derivation rule can be defined or modified using the **Define Derivation** button:

1. **none**: This attribute has no derivation. Previously defined derivation will be removed.
2. **arithmetic**: An **Arithmetic Expression Definition** window (Figure 20) is brought up.
3. **aggregate**: An **Aggregate Function Definition** window (Figure 21) is brought up.
4. **composition**: An **Attribute Composition Definition** window (Figure 22) is brought up.
5. **inverse**: An **Attribute Inverse Definition** window (Figure 18) is brought up.
6. **subvalue**: An **Attribute Subvalue Definition** window (Figure 23) is brought up.
7. **union**: An **Attribute Union Definition** window (Figure 24) is brought up.

A component attribute can have at most one derivation. If the composite attribute containing the component attribute is an identifier or is derived using attribute match, then the component attribute cannot have a derivation. After attribute derivation is defined, click **Modify** button to associate the component attribute with the new derivation.

4.10.2 Modify Component Attribute

Attribute names, value classes and derivation rules of component attributes can be modified. Changes are finalized (i.e., recorded as schema updates) using the **Modify** button.

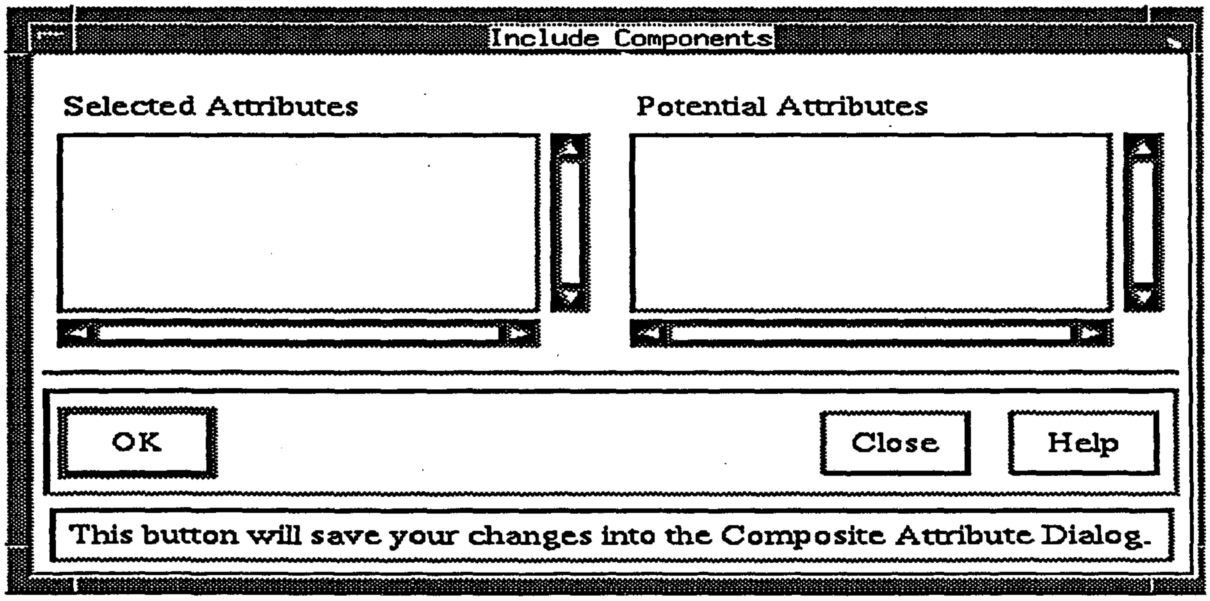
The modification of value class associations and derivation specifications is carried out as described in the subsection **Add a Component Attribute.**

4.10.3 Delete Component Attribute

A component attribute can be deleted using the **Delete** button. Deletion is carried out only after the user confirms the action.

4.11 Include Attributes into a Composite Attribute

Component attributes of a composite attribute can be defined one by one using the **Component Attribute Definition** window. Alternatively, existing simple attributes of the target object or protocol class can be included as components of a composite attribute using the **Include Components** window (Figure 11).



This button will save your changes into the Composite Attribute Dialog.

FIGURE 11. **Include Components** Window

The **Selected Attributes** Listbox and the **Potential Attributes** Listbox are complementary. An existing simple attribute of the current class is displayed in one (and only one) of the two listboxes.

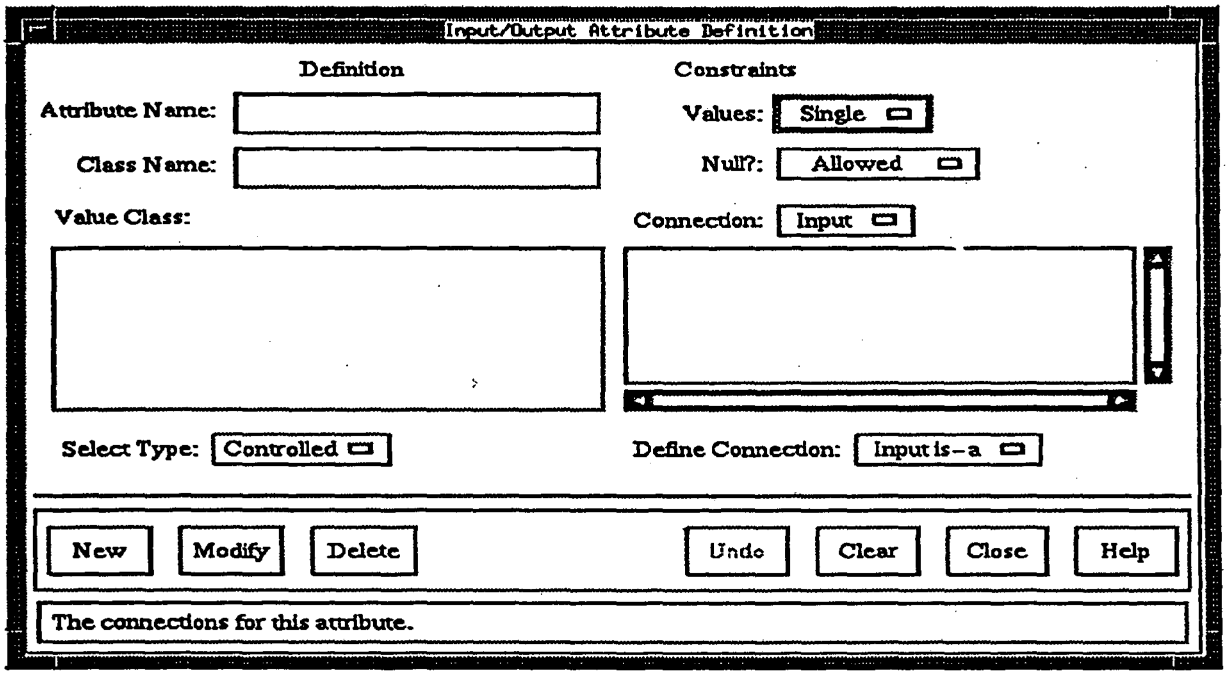
Selected Attributes Listbox contains all the attributes that will be included as components of the composite attribute.

Initially, the **Selected Attributes** Listbox is empty, and the **Potential Attributes** Listbox contains all the simple attributes defined for the current class. When an attribute name is selected in the **Potential Attributes** Listbox, this attribute is moved from the **Potential Attributes** Listbox to the **Selected Attributes** Listbox, and vice versa.

After all the included attributes are properly selected, the **OK** button must be used in order to record the change to the schema.

4.12 Define Input/Output Attribute

Input/output attributes are associated only with protocol classes. Every input/output attribute has an attribute name, attribute constraints, is associated with a value class, and has an input/output connection specification.



The connections for this attribute.

FIGURE 12. **Input/Output Attribute Definition** Window

4.12.1 Add Input/Output Attribute

The name of the current protocol class is displayed in the **Class Name** area. A new attribute must have an attribute name, a value class, defined constraints and an input/output connection specification before being associated with the protocol class.

An input/output attribute must have a distinct non-null name within a protocol class. Such a name is in lower case letters and cannot exceed 32 characters.

The **Single** option of the **Values** button indicates that the attribute can have only one value associated with it; if the attribute can be associated with sets of values, then the **Multiple** *option* must be selected.

The **Allowed** option of the **Null** button indicates that the attribute can have null values; if the attribute is not allowed to have null values, then the **Not Allowed** option must be selected.

Every attribute must have an associated value class. The value class of the current input/output attribute is listed in the **Value Class** Listbox. The **Select Type** option menu can be used for defining or modifying the value class. An input/output attribute can be associated with one of the following three types of value classes:

1. **Controlled**: An **Attribute Controlled Value Class** window (Figure 14) is brought up.
2. **Primitive**: An **Attribute Primitive Value Class** window (Figure 15) is brought up.
3. **Abstract**: An **Attribute Abstract Value Class** window (Figure 16) is brought up.

When the type of value class is changed, the user is informed that the previously defined value class is destroyed.

After all the necessary information is specified, the input/output attribute is associated with the protocol class using the **New** button.

An input/output attribute can have an associated connection statement. In order to define a correct input/output connection, the protocol expansion of the current protocol class should be first defined and saved, and the input/output attribute should be (correctly) associated with a value class.

Input/output connections are listed in the **Connection** box. Connections are defined or modified using the **Define Connection** option menu:

1. **Input is-a**: An **Input/Output Is-a Definition** window (Figure 25) is brought up.
2. **Input from**: An **Input From Definition** window (Figure 26) is brought up.
3. **Output is-a**: An **Input/Output Is-a Definition** window (Figure 25) is brought up.

An attribute can be either an input attribute or an output attribute, but not both. If an attribute is changed from an input attribute to an output attribute, then the from-statement will be lost and the user is notified of this loss.

After the input/output connection is specified, click **Modify** button to associate the connection to the current attribute.

4.12.2 Modify Input/Output Attribute

Attribute names, value classes, constraints and input/output connections of input/output attributes can be modified. Changes are finalized (i.e., recorded as schema updates) using the **Modify** button.

The modification of attribute constraints, value classes, and connection specifications is carried out as described in the subsection **Add an Input/Output Attribute.**

4.12.3 Delete Input/Output Attribute

An input/output attribute can be deleted using the **Delete** button. Deletion is carried out only after the user confirms the action.

4.13 Define Controlled Value Class

A controlled value class is a primitive value class with enumerated atomic values or ranges. For example, a controlled value class COLOR with Character String type has values: *red*, *yellow* and *green.* Another controlled value class AGE_GROUP with Numeric Constant type has values 20-55.

A controlled value class can be defined or modified using the **Controlled Value Class/Value** window (Figure 13).

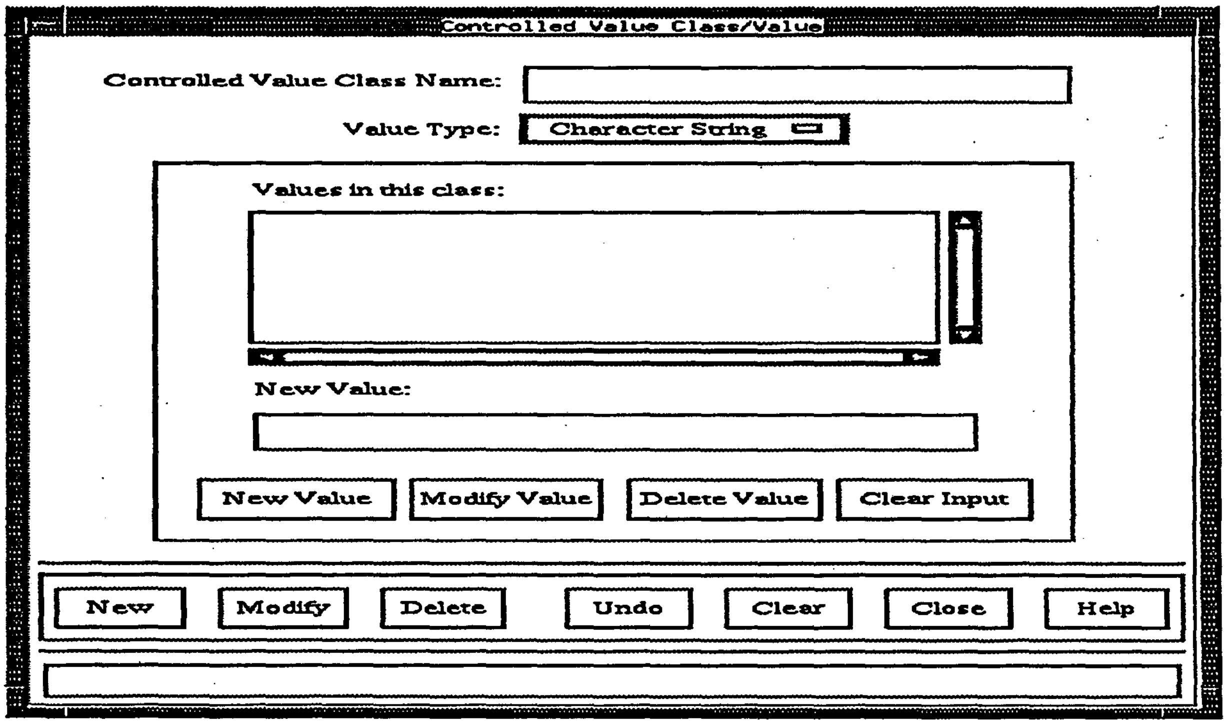


FIGURE 13. **Controlled Value Class/Value** Window

4.13.1 Add Controlled Value Class

Before being added, a controlled value class must be associated with a distinct non-null (unique) class name. This name must be in upper case letters and cannot exceed 32 characters.

A controlled value class can have Character String or Numeric Constant value type. The value type determines the data type of user's input. Thus, an input value *n* is considered as a number if the value type is Numeric Constant, and is considered as a string "*n*" if the value type is Character String.

The values of a controlled value class are added one by one as follows: first the new value is entered in the **New Value** area; then the insertion is finalized using the **New Value** button. The entered new value is listed in the **Values in this class** Listbox.

Values in a controlled value class with Character String value type are distinct character strings that do not exceed 80 characters. Values in a controlled value class with Numeric Constant value type are numbers or ranges. A range is represented as: *a* - *b*, where *a* and *b* are both numbers. Negative numbers are enclosed in parentheses. For example, (-5) - 10 is a range with lower bound -5 and upper bound 10.

A value is modified or removed from the current value class as follows:

1. the value is first selected by clicking on it in the **Values in this class** Listbox;
2. the selected value is highlighted and copied to the **New Value** area;
3. if the value is modified then
   (a) the value is modified in the **New Value** area, and the change is finalized using the **Modify Value** button; the highlighted value in the listbox is replaced by the new value;
   (b) if the **New Value** button is used instead of the **Modify Value** button in the previous step, then the edited value is inserted as a new value into the controlled value class;
4. the highlighted value in the listbox can be removed by using the **Delete Value** button.

The **Clear Input** button clears the **New Value** area.

After all the values of a controlled value class are defined, the controlled value class is added to the current schema using the **New** button.

4.13.2 Modify Controlled Value Class

An existing controlled value class can be modified using the **Modify** button. All the values in a controlled value class as well as the name of the controlled value class can be modified. The procedure to add, modify or delete a value in an existing controlled value class is the same as the value modification procedure described in **Add a Controlled Value Class.**

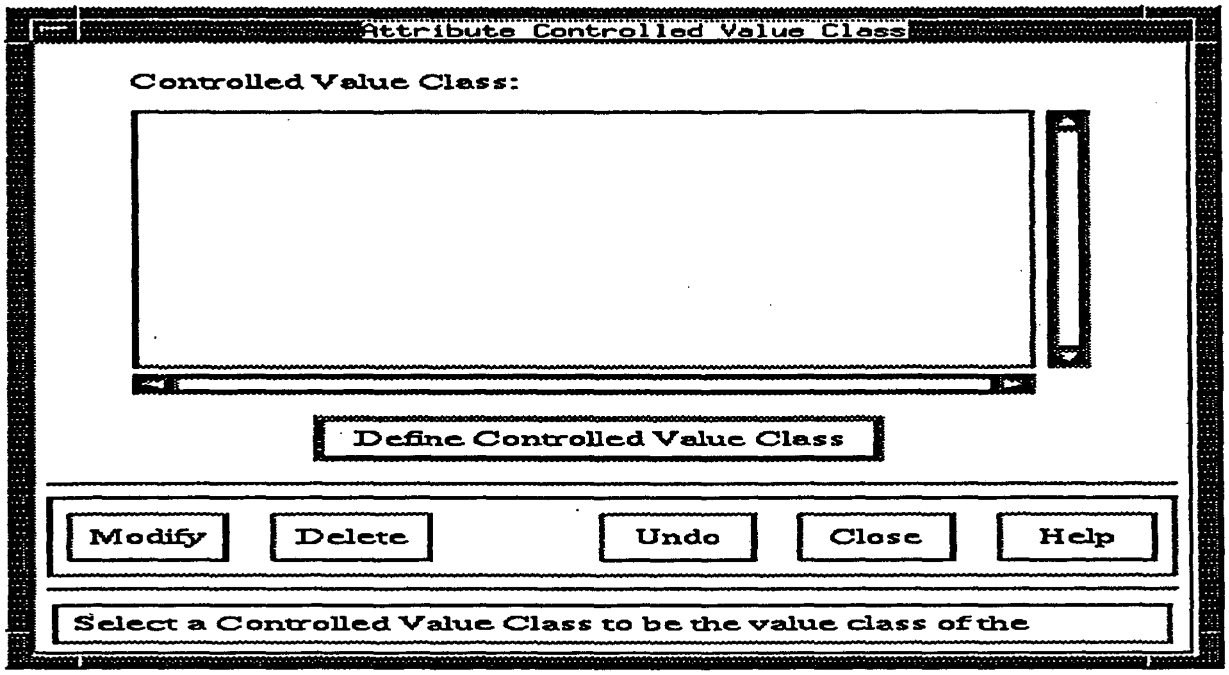
The value type of a controlled value class can also be modified. It is always possible to convert Numeric Constant type to Character String type. All the numbers and ranges defined in the controlled value class are converted into strings. A controlled value class with Character String value type can be converted into Numeric Constant type only when all the values defined in this class can be converted into numbers and/or ranges; otherwise, an error message will be issued and the value type remains Character String.

4.13.3 Delete Controlled Value Class

An existing controlled value class can be deleted using the **Delete** button. Deletion is carried out only after the user confirms the action.

4.14 Select Attribute Controlled Value Class

All the predefined controlled value classes together with their values are listed in the **Controlled**


Attribute Controlled Value Class
Controlled Value Class:
Define Controlled Value Class
Modify
Delete
Undo
Close
Help

Select a Controlled Value Class to be the value class of the

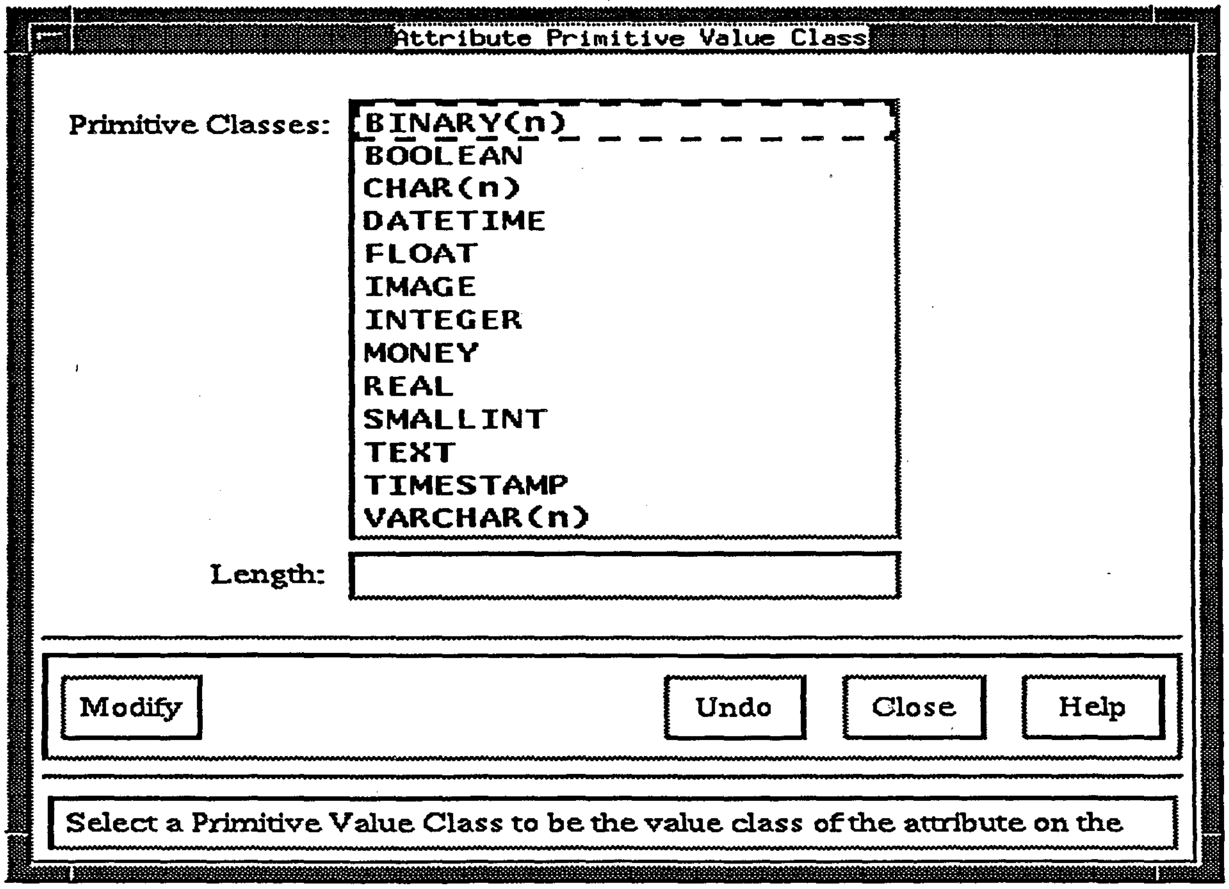
FIGURE 14. **Attribute Controlled Value Class** Window

Value Class Listbox. For example, if there is a controlled value class called COLOR with three values: *red*, *yellow* and *green*, then it is displayed as: COLOR: {"*red* ","*yellow* ","*green*"} (for the definition and modification of controlled value class see section **Define Controlled Value Class**).

The highlighted value class in the listbox will be the value class associated with current attribute. For adding or modifying a controlled value class of an attribute, the desired value class name must be first selected in the listbox (the class name will be highlighted), and then the **Modify** button must be used.

If the attribute is associated with a new (not previously defined) controlled value class, then the **Define Controlled Value Class** button must be used first in order to define the controlled value class. **The Define Controlled Value Class** button brings up the **Controlled Value Class/Value** window (Figure 13) for the definition of a new controlled value class. After the new controlled value class is added, the name of this new class is listed and highlighted in the **Controlled Value Class** Listbox. The **Modify** button is then used in order to associate the attribute with this value class.

The value class of an attribute is deleted using the **Delete** button. A controlled value class is preserved in the current schema after such a deletion.



Select a Primitive Value Class to be the value class of the attribute on the

FIGURE 15. **Attribute Primitive Value Class** Window

4.15 Select Primitive Value Class

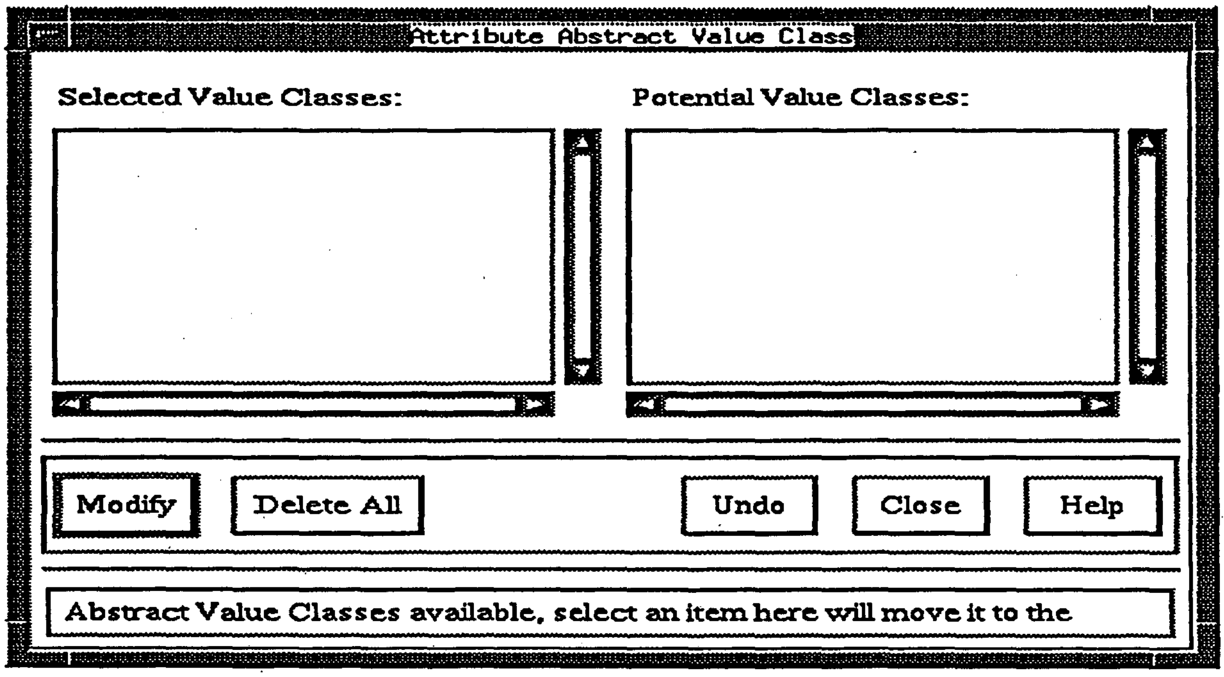
In order to allow selecting a primitive value class, the **Primitive Classes** Listbox lists all the system defined primitive classes.

A primitive value class can be selected by clicking on it in the listbox. The selected primitive value class will be highlighted. If this primitive value class has "(n)" at the end, then the attribute length must be specified; a length is a positive integer.

After a value class is selected (and a length defined), the attribute is associated with the value class using the **Modify** button.

4.16 Select Abstract Value Class

An attribute can be associated with one or several abstract value classes (i.e., value classes that are defined as object classes).



Abstract Value Classes available, select an item here will move it to the

FIGURE 16. **Attribute Abstract Value Class** Window

The **Selected Value Classes** Listbox and **Potential Value Classes** Listbox are complementary. The name of an OPM class (including the current one) is listed in exactly one of these listboxes. The **Selected Value Classes** Listbox contains the value classes selected for the attribute.

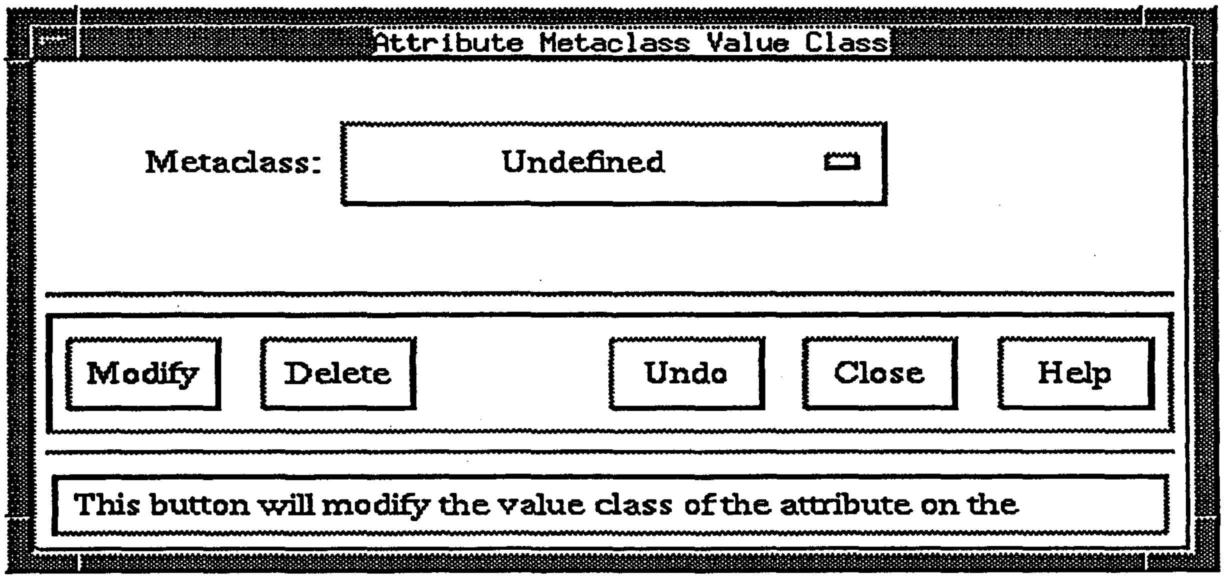
A class selected (clicked on) in the **Selected Value Classes** Listbox, is moved from the **Selected Value Classes** Listbox to the **Potential Value Classes** Listbox, and vice versa.

The **Delete All** button can be used to clear the **Selected Value Classes** Listbox, and thus move all the classes to the **Potential Value Classes** Listbox.

The selected value classes are associated with the attribute using the **Modify** button.

4.17 Select Metaclass Value Class

There are two Metaclasses in OPM: OBJECT_CLASSES and PROTOCOL_CLASSES.



This button will modify the value class of the attribute on the

FIGURE 17. **Attribute Metaclass Value Class** Window

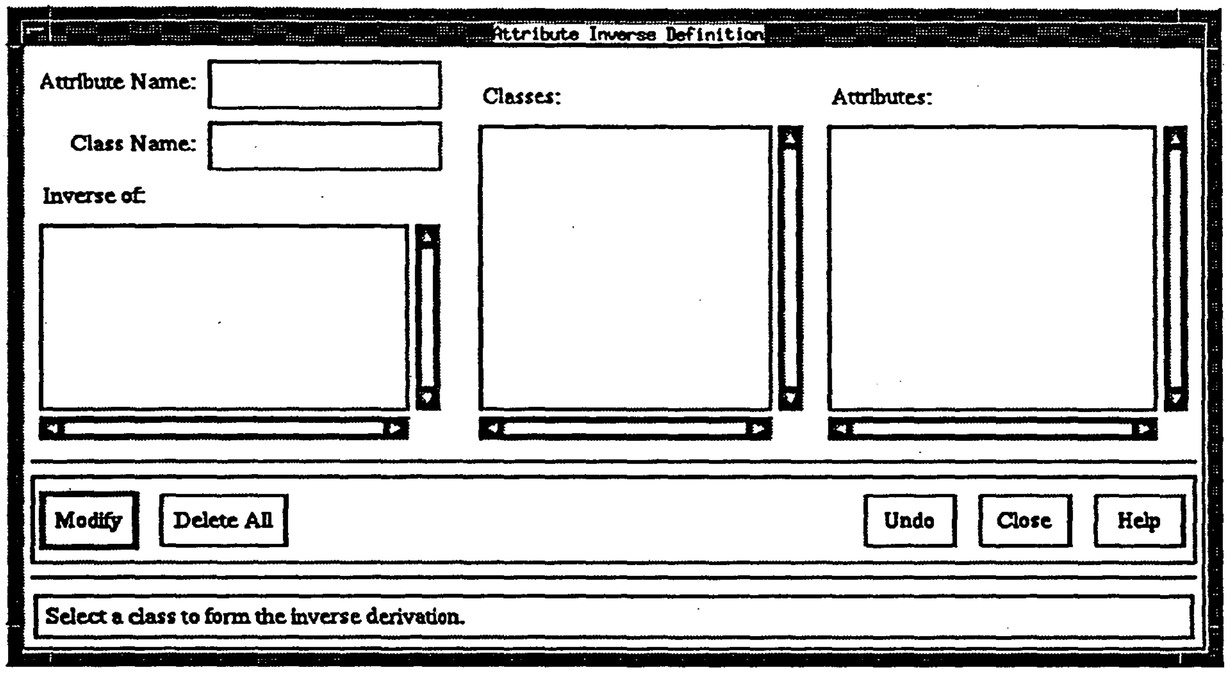
A user can select a metaclass using the **Metaclass** option menu; the default option is *Undefined*. The selected (option) metaclass is highlighted. The highlighted value class in the **Metaclass** option menu is associated with the attribute using the **Modify** button.

A metaclass can be deleted as the value class of an attribute using the **Delete** button.

4.18 Define Attribute Inverse Derivation

The current class name and attribute name are displayed in the **Attribute Name** and **Class Name** areas, respectively (at the top of the window). An attribute can be specified as inverse of multiple attributes; all these attributes are listed in the **Inverse of** Listbox.The object classes that are defined as value classes of the current attribute are listed in the **Classes** Listbox. When a class name in the **Classes** Listbox is selected (clicked on), the (simple or component) attributes of that class will be listed in the **Attributes** Listbox.

In order to define an attribute A as an inverse of the current attribute, class O_A of A must be first



Select a class to form the inverse derivation.

FIGURE 18. **Attribute Inverse Definition** Window

selected. As a result, the attributes of O_A are listed in the **Attributes** Listbox. Subsequently, the name of attribute A is selected in the **Attributes** Listbox and the name of the select attribute prefixed by the name of its class (i.e., $O_A.A$) is inserted into the **Inverse of** Listbox

An attribute inverse definition can be removed by selecting (clicking on) the attribute name in the **Inverse of** Listbox. In order to remove all attribute inverse definitions (i.e., clear the **Inverse of** Listbox), the **Delete All** button is used.

After the inverse derivation of the current attribute is defined, the derivation is associated with the attribute using the **Modify** button. A message will be brought up to ask whether you want to make this attribute an inverse of corresponding attribute(s) of the selected class(es). Click **Yes** will add the current attribute to the attribute inverse derivations of all the attributes specified in the **Inverse of** Listbox.

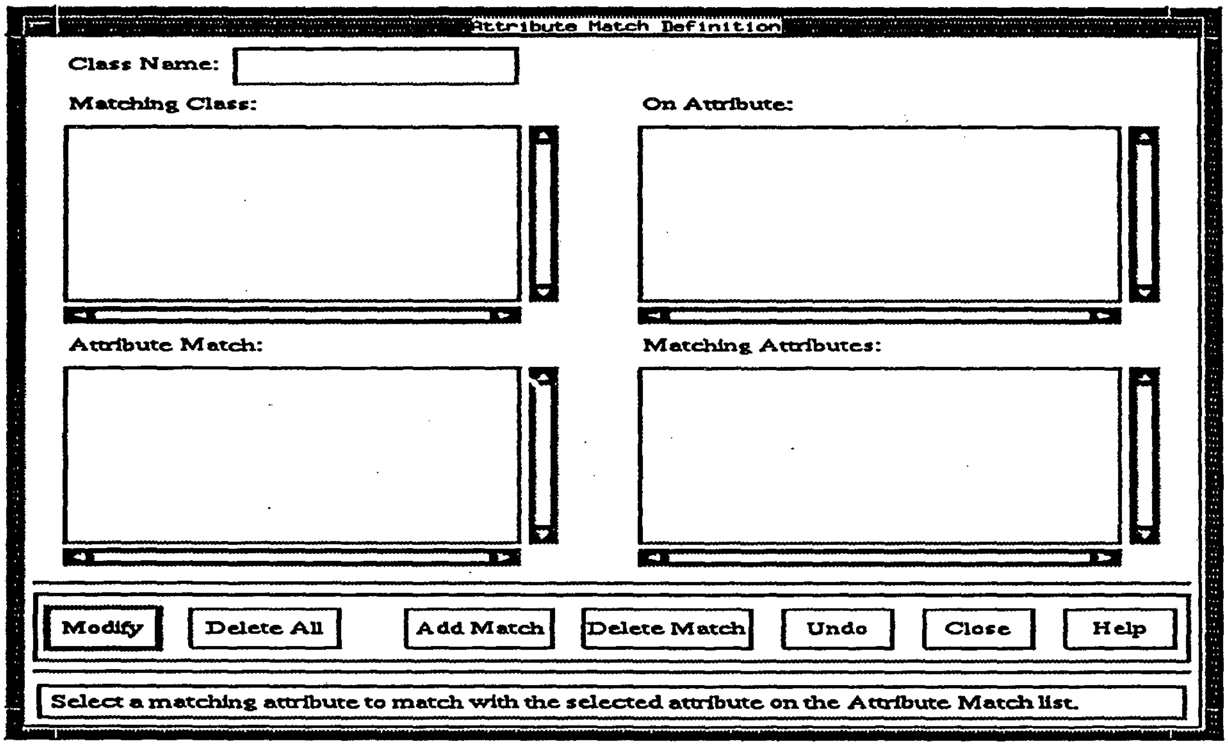
The following constraints must be satisfied when an attribute inverse is defined:

1. The value classes of the attributes defined as inverses of the current attribute must contain (or be equal to) the object class of the current attribute.
2. Attributes defined as inverses must either be non-derived, or defined as inverses of the current attribute.
3. Only one attribute can be selected from each class.

4. If the current attribute is associated with the union of value classes V_1 or ... or V_n, then only simple attributes of these classes $V_1, \ldots, V_n$ are listed in the **Attributes** Listbox (no composite or component attributes).

4.19 Define Attribute Matching Derivation

The name of the current class is displayed in the **Class Name** area at the top of the window. The name of the attribute that is to be matched (i.e., the attribute is associated with a match derivation) is listed in the **Attribute Match** Listbox as follows: if the attribute is a simple attribute, then its name is listed in the listbox; if the attribute is a composite attribute, then all the names of the component attributes are listed in the listbox.



Select a matching attribute to match with the selected attribute on the Attribute Match list.

FIGURE 19. **Attribute Match Definition** Window

The **Matching Class** Listbox lists the classes that can be used in the matching derivation; how to determine whether a class can be used or not in the matching derivation of the current attribute is explained in more detail later in this subsection.

The matching is defined as follows:

1. A class name is selected in the **Matching Class** Listbox, and as a result the components of a composite attribute of that class satisfying the matching constraints are listed in the **On Attribute** Listbox.

2. Next, a match on attribute B_1 is selected in the **On Attribute** Listbox, and as a result the component attributes that belong to the same composite attribute as B_1, except B_1, are listed in the **Matching Attributes** Listbox.

3. An attribute is selected in the **Attribute Match** Listbox (listing the simple or components attributes to be matched) and then its matching attribute is selected in the **Matching Attributes** Listbox;

4. The match association in the previous step is defined using the **Add Match** button. As a result, the selected matching attribute is *included* in parentheses in the **Attribute Match** Listbox appended after the matched attribute, and is removed from **Matching Attributes** Listbox. (For example, if attribute A_1 matches A_2, then A_1 (A_2) will replace the original item A_1 in the **Attribute Match** Listbox; A_2 is removed from **Matching Attributes** Listbox.) If an attribute already has a matching attribute, then **Add Match** replaces the matching attribute.

The matching attribute of an attribute selected in the **Attribute Match** Listbox can be removed using the **Delete Match** button. A removed matching attribute is returned to the **Matching Attributes** Listbox.

All the matching attributes can be removed from all attributes listed in the **Attribute Match** Listbox using the **Delete All** button. The removed matching attributes are returned to the **Matching Attributes** Listbox.

Note that after a matching is specified, the **Matching Class** Listbox cannot be changed. This listbox can be changed only if all matching attributes are removed (using **Delete All**).

After the matching derivation is correctly defined, the match derivation is associated with the current attribute using the **Modify** button.

The attribute matching derivation must satisfy the following additional constraints:

1. In the **Matching Class** Listbox only classes that have a *composite attribute* that contain a *component attribute* whose value class includes the *class of the current attribute* are listed.

2. A simple attribute can match only a composite attribute with two components. A composite attribute A with n components can match only another composite attribute B with $(n+1)$ components. Consequently:

(a) For a simple attribute match, only component attributes of binary composite attributes are listed in the **On Attribute** Listbox.

(b) For a composite attribute match, if the attribute is an n-ary composite attribute, then only component attributes of (n+1)-ary composite attributes are listed in the **On Attribute** Listbox.

3. An attribute *A* can match an attribute *B* from the **Attribute Match** listbox only if *A* and *B* have the same value class.

4. The value class of an **On Attribute** must include the current object class. Consequently, in the **On Attribute** Listbox are listed only the names of attributes that are associated with value classes that include the current object class.

An matching example is given immediately below.

An Example:

Let object classes TRANSLATES and GENE be defined as follows:

OBJECT CLASS TRANSLATES:

DESCRIPTION: gene translates protein at some cell
ID: (gene, at_cell, protein)
ATTRIBUTE (gene, at_cell, protein): (GENE, CELL, PROTEIN) single-valued not null

OBJECT CLASS GENE:

DESCRIPTION: gene
ID: gene_name
ATTRIBUTE gene_name: CHAR(80)
ATTRIBUTE (translate_protein, translate_at): (PROTEIN, CELL)

Suppose a match derivation for the composite attribute (translate_protein, translate_at) of object class GENE is defined as: *match* (protein, at_cell) *of* TRANSLATES *on* gene. Components translate_protein and translate_at are listed in the **Attribute Match** Listbox.

All the (object and protocol) classes that have attributes that satisfy the matching constraints are listed in the **Matching Class** Listbox. Suppose that TRANSLATES is selected in the **Matching Class** Listbox. Since the current composite attribute has two components, only a composite attribute of TRANSLATES consisting of three components can be selected for matching, that is, composite attribute (gene, at_cell, protein).

Among the component attributes of (gene, at_cell, protein) only attribute gene has value class GENE , and therefore only attribute gene is listed in the **On Attribute** Listbox.

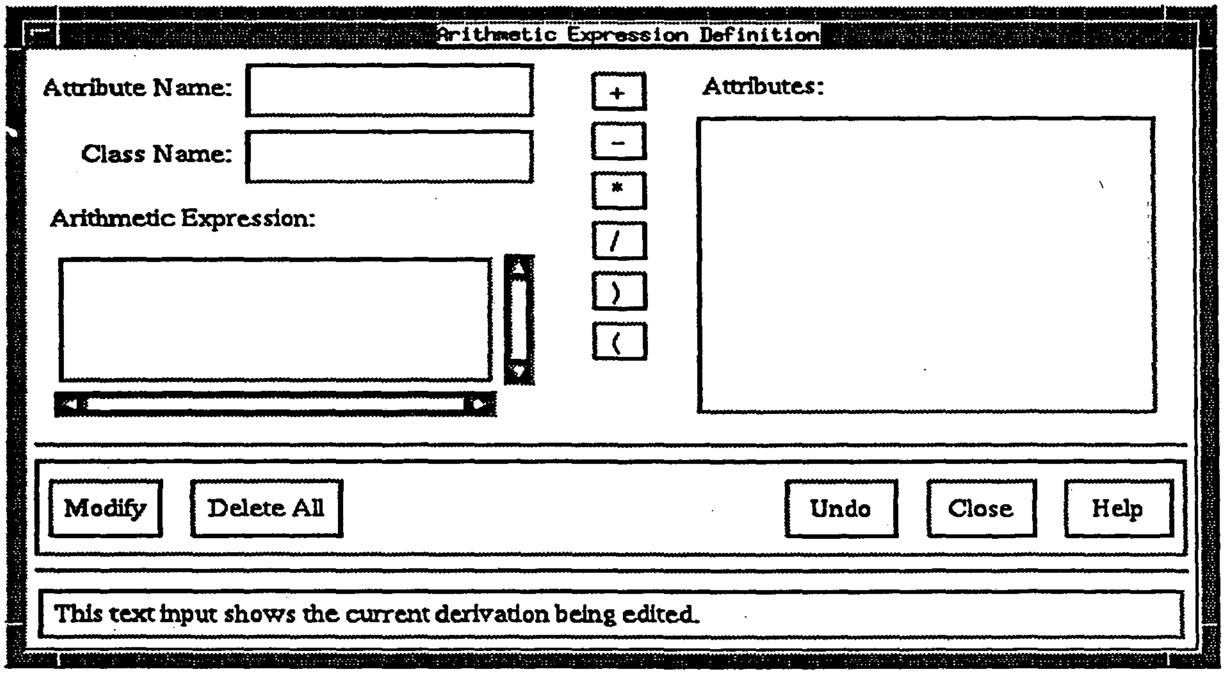
If attribute gene is selected in the **On Attribute** Listbox, then the other two components, at_cell and protein, are listed in the **Matching Attributes** Listbox.

Next, translate_protein is selected in **Attribute Match** Listbox, and protein is selected in the **Matching Attribute** Listbox; using the **Add Match** button the attribute name translate_protein in **Attribute Match** Listbox is replaced by: translate_protein (protein), and attribute protein is then removed from the **Matching Attributes** Listbox.

The same procedure is repeated in order to match translate_at and at_cell.

4.20 Define Arithmetic Expression Derivation

The current class name and attribute name are displayed in the **Class Name** and **Attribute Name** areas, respectively, at the top of the window. The arithmetic expression derivation to be associated with the attribute is displayed in the **Arithmetic Expression** working area. The arithmetic expression can be



This text input shows the current derivation being edited.

FIGURE 20. **Arithmetic Expression Definition** Window

directly edited in the **Arithmetic Expression** working area, or can be expressed using the **Attributes** Listbox and the operator buttons.

All the single-valued simple and component attributes of the current class (except the current attribute), associated with an INTEGER, SMALLINT, REAL, FLOAT, or MONEY value class are listed in the **Attributes** Listbox. Only these attributes can be used in the arithmetic derivation.

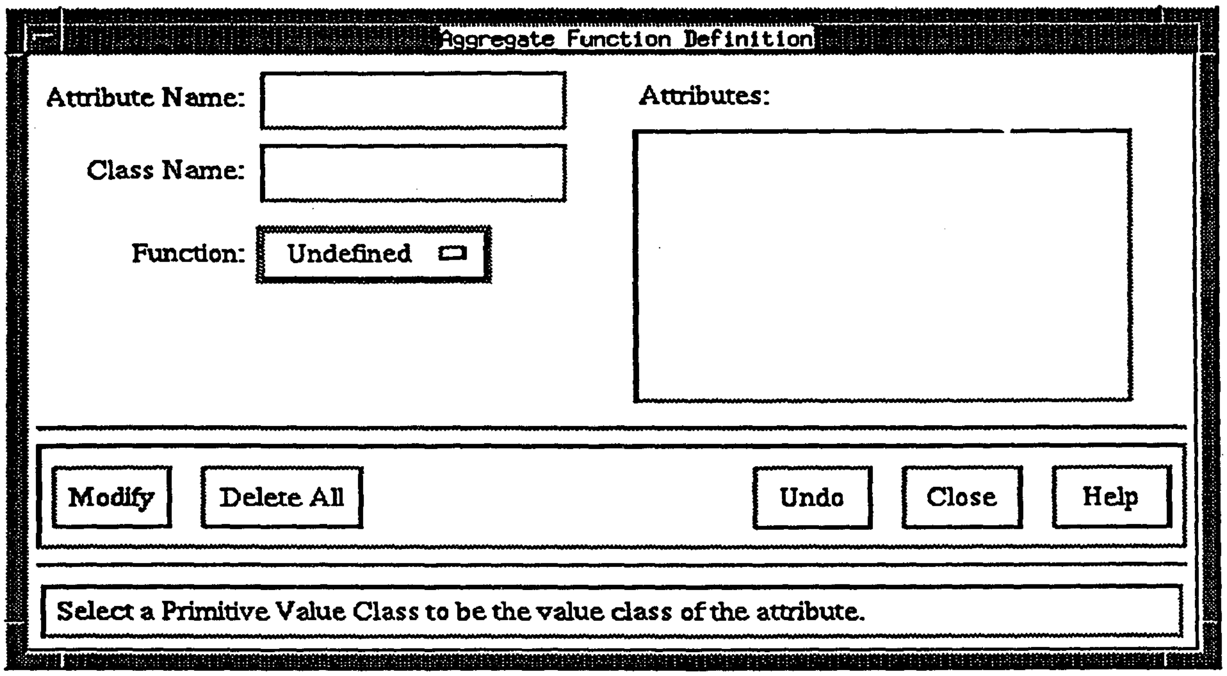
An attribute name selected in the **Attributes** Listbox is inserted into the **Arithmetic Expression** working area. Selecting (clicking on) a special operator button (+, -, *, /, (,)) results in inserting the corresponding symbol into the **Arithmetic Expression** working area as well.

The **Arithmetic Expression** working area can be cleared using the **Delete All** button.

After the arithmetic expression derivation has been defined, the attribute is associated with the arithmetic expression derivation using the **Modify** button.

4.21 Define Aggregate Function Derivation

The current class name and attribute name are displayed in the **Class Name** and **Attribute Name** areas, respectively, at the top of the window. An aggregate function derivation consists of an aggregate function (count, min, max, sum, average) and an attribute name. All the multi-valued (simple or component) attributes of the current class are listed in the **Attributes** Listbox.



Select a Primitive Value Class to be the value class of the attribute.

FIGURE 21. **Aggregate Function Definition** Window

First, one of the functions in the **Function** option menu (**count, min, max, sum, average**) must be selected. Then, an attribute name listed in the **Attributes** Listbox is selected (clicked on); the selected attribute is highlighted.

If the selected function is **min**, **max**, **sum** or **average**, then only multi-valued attributes associated with an INTEGER, SMALLINT, REAL, FLOAT, or MONEY value class are listed in the **Attributes** Listbox.

The **Function** selection and the attribute selection in the **Attributes** Listbox can be cleared using the **Delete All** button.

After the aggregate function derivation has been defined, the attribute is associated with the derivation using the **Modify** button.

4.22 Define Attribute Composition Derivation

A simple or component attribute C can be derived as: $A_1.A_2. \ldots .A_n$ ($n \geq 2$), where A_1 is an attribute associated with the current class, A_2 is an attribute associated with the value class(es) of A_1, etc. Attributes $A_1, A_2, \ldots, A_n$ are all simple or component attributes; they are either non-derived or derived by inverse derivation. Attribute A_n can either be an abstract attribute or a primitive attribute.

Attribute composition derivation is defined in the **Attribute Composition Definition** window (Figure 22).

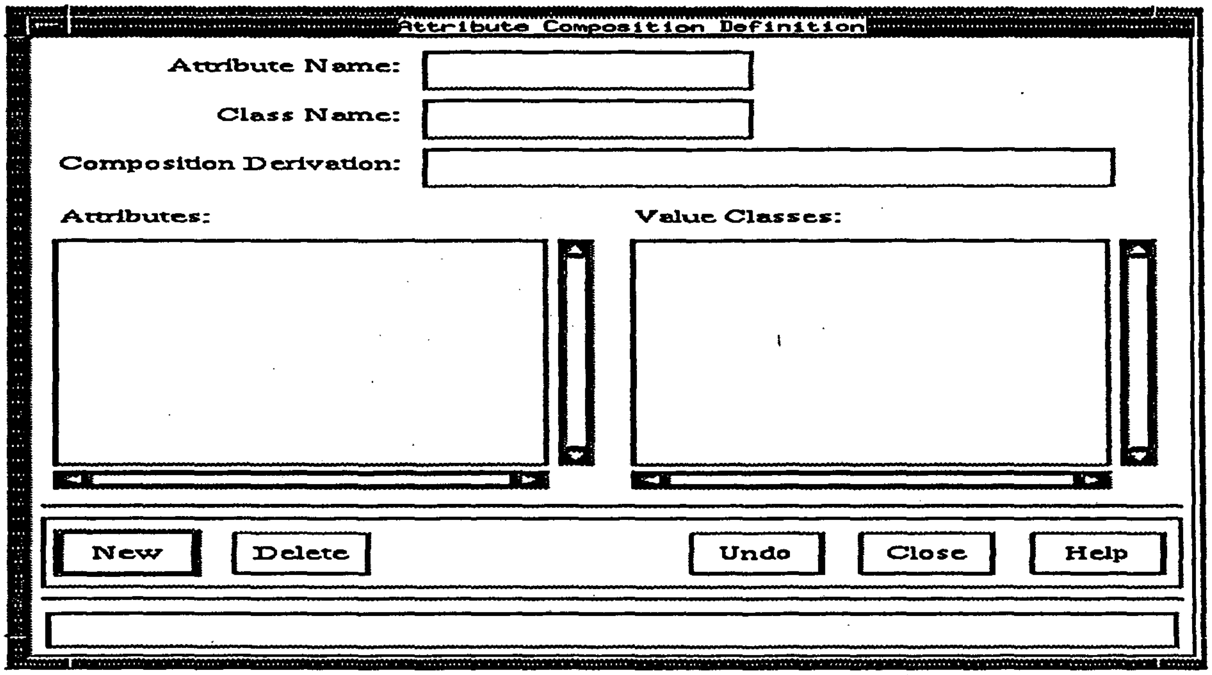


FIGURE 22. **Attribute Composition Definition** Window

The current attribute name and class name are displayed at the top of the window in the **Attribute Name** area and **Class Name** area, respectively. The attribute composition derivation is displayed in the

Composition Derivation area.

In the beginning, all local (i.e., not inherited) non-derived or inverse derived, simple or component attributes (not composite attributes) associated with the current class, O_i, except the current attribute are displayed in the **Attributes** Listbox.

When an attribute is selected in the **Attributes** Listbox, this attribute is highlighted and displayed in the **Composition Derivation** area. The value class(es) of this selected attribute will be listed in the **Value Classes** Listbox. This **Value Classes** Listbox is for display only; it is non-selectable.

In the case of controlled value class, primitive value class or metaclass value class, the **Attributes** Listbox remains unchanged.

In the case of an abstract value class:

- if the value class consists of a single abstract object class, O_j, then all locally simple or component attributes of O_j are displayed in the **Attributes** Listbox.
- if the value class consists of a union of the object classes, O_{j_1} or ... or O_{j_m}, then local simple or component attributes that are associated with O_{j_1} and ... and O_{j_m} are displayed in the **Attributes** Listbox; for each such attribute, A, $O_{j_1}.A$, ... , and $O_{j_m}.A$ must be associated same value class.

The selection in the **Attributes** Listbox can be repeated. The selected attribute name is appended at the end in **Composition Derivation** area. (A dot "." is automatically added between any two attribute names.) The value class of a newly selected attribute is again displayed in the **Value Classes** Listbox.

The definition of composition derivation stops either at a non-abstract attribute, or when the user ends selecting attributes.

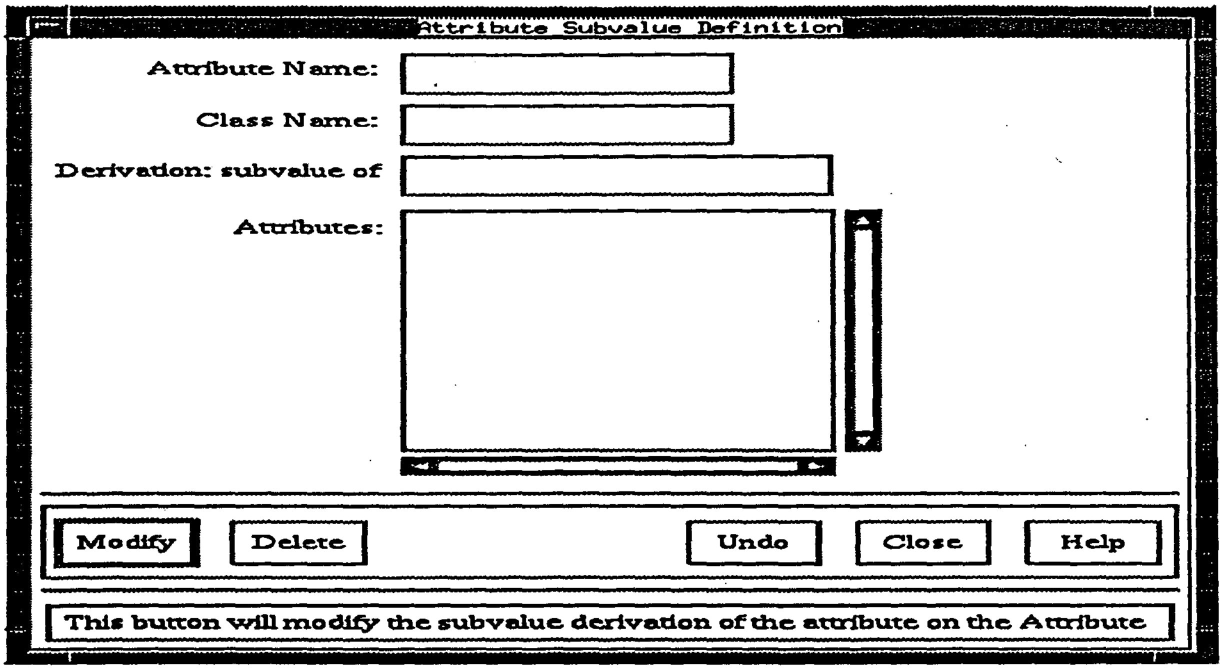
After the derivation has been defined, the attribute is associated with the new derivation using the **New** button. The **Delete** button removes the composition derivation.

4.23 Define Attribute Subvalue Derivation

An attribute A_1 of an object or protocol class O_i can be defined as: subvalue of A_2, if the value class of A_1 is a subclass or subset of the value class of A_2. Attribute subvalue derivation is defined in the **Attribute Subvalue Definition** Window (Figure 23).

The current class name and attribute name are displayed in the **Class Name** area and **Attribute Name** area, respectively. **Derivation: subvalue of** area displays the attribute subvalue derivation.

Suppose A is the current attribute. A simple or component attribute of the current class (except for the current attribute), B, is listed in the **Attributes** Listbox if:



This button will modify the subvalue derivation of the attribute on the Attribute

FIGURE 23. **Attribute Subvalue Definition** Window

1.attribute ***B*** is associated with a value class consisting of a single class, O_B , attribute *A* is associated with a value class consisting of a single class, O_A, and O_B is an immediate or transitive superclass of O_A;

2.attribute ***B*** is associated with a value class consisting of a union of value classes, O_{B_1} or ... or O_{B_n} . Attribute *A* is associated with a value class consisting of a single or a union of classes, O_A, and O_A is contained in O_{B_1} or ... or O_{B_n} .

After an attribute is selected in the **Attributes** Listbox, the selected attribute name is displayed in the **Derivation: subvalue of** area. The attribute is associated with the derivation using the **Modify** button.

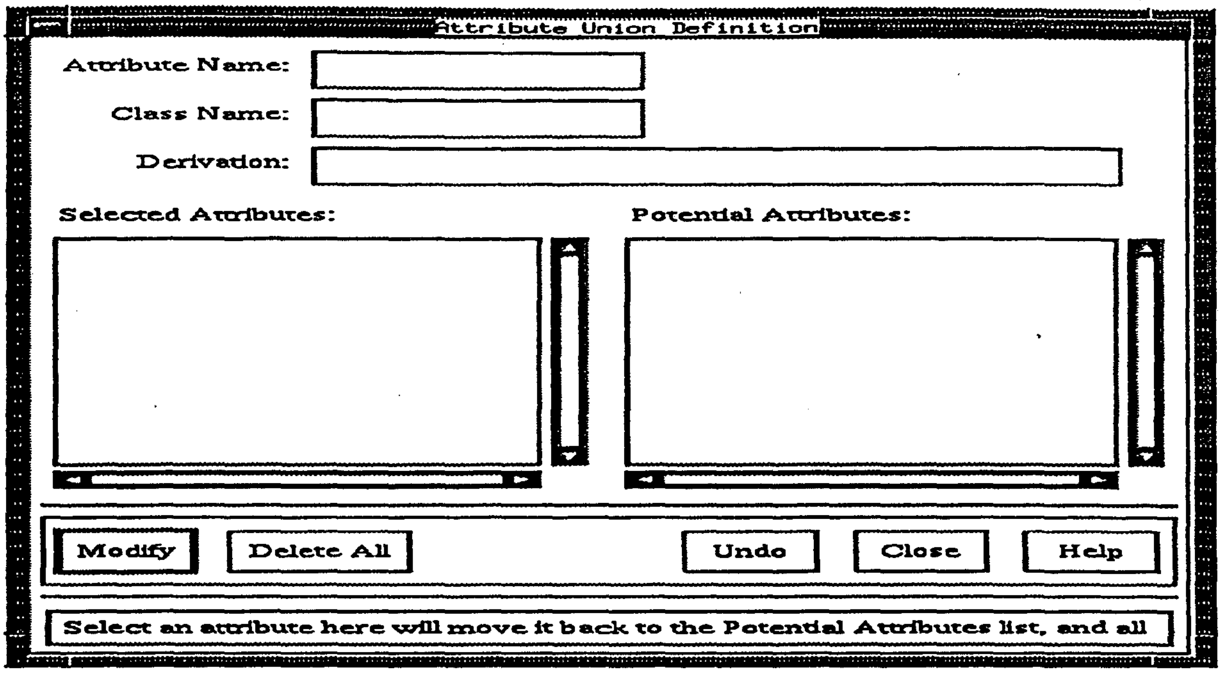
Delete button removes the subvalue derivation.

4.24 Define Attribute Union Derivation

Attribute union derivation is defined in the **Attribute Union Definition** window (Figure 24).

The current class name and attribute name are displayed in the **Class Name** area and **Attribute Name** area, respectively. **Derivation** area displays the attribute union derivation.

Selected Attributes Listbox and **Potential Attributes** Listbox are complementary. Suppose that



Select an attribute here will move it back to the Potential Attributes list, and all

FIGURE 24. **Attribute Union Definition** Window

the current attribute is ***B***. Simple or component attributes (except ***B***) associated with the current class and that have value classes that are subsets of (or equal to) the value class of ***B*** are displayed in one (and only one) of the two listboxes. **Selected Attributes** Listbox contains the attributes that will be included in the derivation.

An attribute that is selected (clicked on) in the **Potential Attributes** Listbox is moved from the **Potential Attributes** Listbox to the **Selected Attributes** Listbox, and vice versa.

Delete All button removes the attribute union derivation; the **Derivation** area and **Selected Attributes** Listbox are cleared, where the attributes in **Selected Attributes** Listbox are moved to the **Potential Attributes** Listbox.

After the union derivation has been defined, the attribute is associated with the derivation using the **Modify** button.

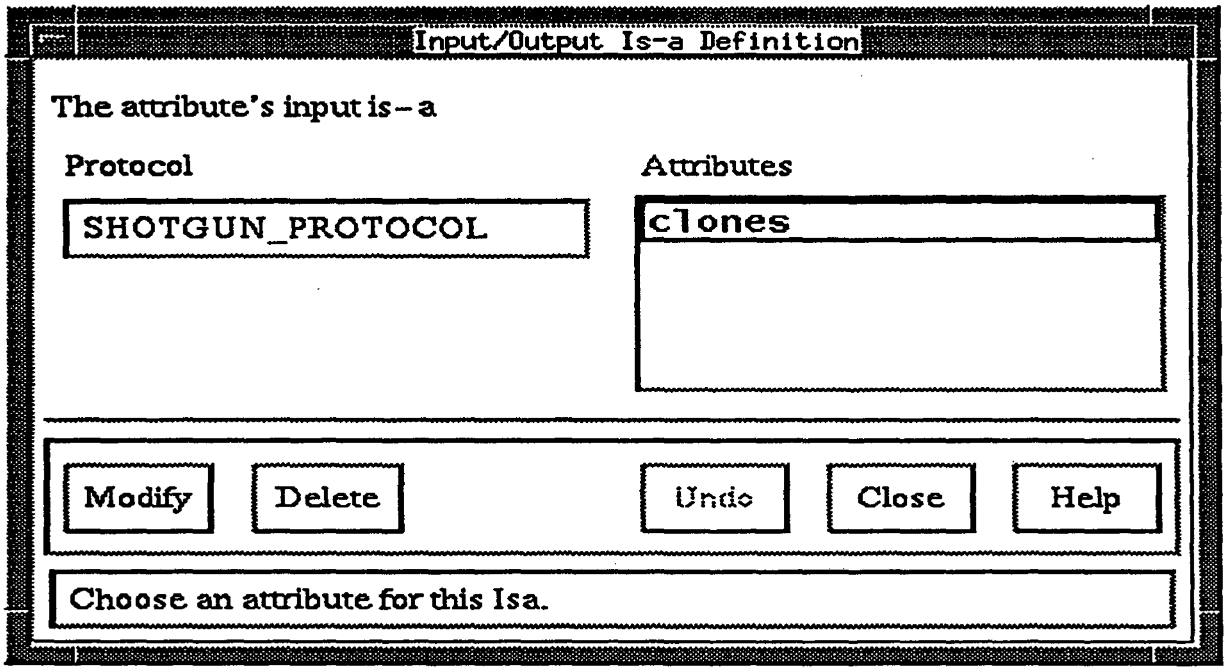
4.25 Define Input/Output Attribute Is-a Connection

This attribute is specified as associated with an input/output is-a connection; the type of connection (*input is-a* or *output is-a*) is displayed at the top of the window.

If the protocol expansion of a protocol class P_1 contains another protocol class P_2, then protocol P_1 is a direct *generic* (higher-level) protocol of P_2. A protocol class can only have one direct generic protocol. An input (output) attribute cannot have an input (output) is-a statement unless the current protocol class has a generic protocol. If an input attribute A_1 of protocol class P_2 *is-a* $P_1.A_1$, then P_1 must be the direct generic protocol of P_2, and A_1 must be an input attribute of P_1. A similar constraint applies to output attribute with *is-a* statement.

The **Protocol** area displays the name of the direct generic protocol of the current protocol. Input attributes or output attributes (depending on whether the attribute is an input or an output attribute) of the generic protocol are listed in the **Attributes** Listbox. The input (output) is-a connection is specified by selecting an attribute from the **Attributes** Listbox; the selected attribute is highlighted in the listbox.

The input (output) is-a connection is removed by using the **Delete** button; the highlight for the pre-



The attribute's input is – a

SHOTGUN_PROTOCOL

clones

Choose an attribute for this Isa.

FIGURE 25. **Input/Output Is-a Definition** Window

viously selected attribute in the **Attributes** Listbox removed.

After the input/output attribute connection has been defined, the attribute is associated with the connection using the **Modify** button.

4.26 Define Input Attribute From Connection

An input attribute *from* connection must satisfy the following constraints: if attribute A of protocol P_i is an input attribute specified as *input from* P_j *via* $B_1, B_2, \ldots, B_n$, then:

1. if P_i is mentioned in the protocol expansion of a protocol class P_k, then P_j must also be mentioned in the same protocol expansion, and P_j must immediately precede P_i;
2. B_1 must be an output attribute of P_j;
3. For every B_m, $2 \leq m \leq n$: B_m is an attribute of class O_m, where O_m is a value class of attribute B_{m-1}.

4.26.1 Add Attribute Input-From Connection

The **From Protocol** Listbox lists the names of the protocol classes that can be used in the input-from connection statement of the current attribute. If the current protocol class, P_i, has been mentioned in a protocol expansion, then the **From Protocol** Listbox lists only the names of the protocol classes that immediately precede P_i in the protocol expansion; otherwise, the **From Protocol** Listbox lists the names of all the protocol classes except P_i.

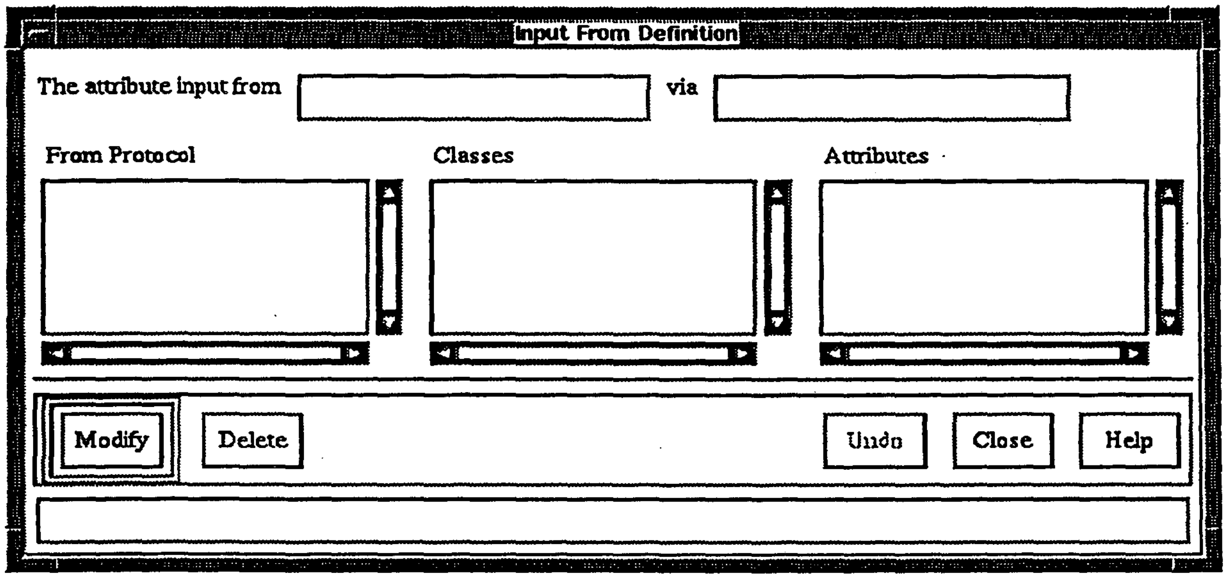


FIGURE 26. **Input From Definition** Window

An input attribute *from* connection is added as follows:

1. A protocol name is selected (clicked on) in the **From Protocol** Listbox; the selected protocol name is highlighted, and is displayed in **The attribute input from** area. As a result of this selection, the output attributes of the selected protocol class are listed in the **Attributes** Listbox.

2. An attribute, ***B***, is selected in the **Attributes** Listbox. As a result, the name of attribute ***B*** is listed in the **via** part (only the attribute name, not class-name.attribute-name, is listed). Following this selection, the value classes of ***B*** are listed in the **Classes** Listbox with the first value class highlighted (selected by default).
3. A class is selected in the **Classes** Listbox. The attributes of this highlighted class are listed in the **Attributes** Listbox. An attribute name is selected in **Attributes** Listbox. As a result of this selection, the name of the selected attribute is appended to the list of attributes already in the **via** area; attribute names in the **via** area are separated by commas (,).
4. The selection of a class name in the **Classes** Listbox and of an attribute name in the **Attributes**

Listbox can be repeated until the input attribute *from* connection specification is completed.

After the input attribute from connection has been defined, the attribute is associated with the connection using the **Modify** button.

4.26.2 Modify Attribute Input-From Connection

An input attribute from connection can be modified following a procedure similar to that described in the previous subsection. After the input attribute from connection has been modified, the attribute is associated with the modified connection using the **Modify** button.

4.26.3 Delete Attribute Input-From Connection

Delete button deletes this input-from connection. An input attribute from connection can be deleted using the **Delete** button.

References

[1] Chen, I-Min A., and Markowitz, V.M., The Object-Protocol Model, Lawrence Berkeley Laboratory Technical Report LBL-32738, 1993.

[2] Chen, I-Min A., and Markowitz, V.M., Mapping Object-Protocol Schemas into Extended Entity-Relationship Schemas and Queries, Lawrence Berkeley Laboratory Technical Report LBL-33048 1993.

[3] Markowitz, V.M., Wang, J., Fang, W. SDT 6.1.. A Schema Definition and Translation Tool for Extended Entity-Relationship Schemas, Lawrence Berkeley Laboratory Technical Report LBL-27843, 1993.

[4] Markowitz, V.M., and Shoshani, A., Object Queries over Relational Databases: Language, Implementation, and Applications, *Proceedings of the 9th International Conference on Data Engineering*, 1993.

A The Object-Protocol Model

In the Object-Protocol Model, objects are qualified by attributes and are classified into **object classes**. Certain objects, called protocols, have additional specific characteristics and therefore are classified into **protocol classes**. Each object or protocol class has a distinct **class name**. An OPM **schema** consists of one or several object and/or protocol classes.

Object and protocol class names are classified in OPM into two **system metaclasses** called OBJECT_-CLASSES and PROTOCOL_CLASSES, respectively. OBJECT_CLASSES contains the names of the object classes defined in the current OPM schema, and PROTOCOL_CLASSES contains the names of the protocol classes in the OPM schema. The content of the system metaclasses reflects the status of an OPM schema, and cannot be changed directly by users.

A.1 Attributes

Attributes in OPM are identified by attribute names, take values from value classes, and can be characterized by attribute constraints. All the (local and inherited) attributes associated with an object or protocol class must have distinct names.

An attribute can be simple or composite. A **simple** attribute is assigned an attribute name and is associated with either a single value class or a union of several value classes. A **composite** attribute consists of several **component** attributes enclosed within parentheses. The name of a composite attribute is optional. However, each component attribute must have a distinct name and an associated (single or union) value class. The constraints associated with a composite attribute apply to all the component attributes. Composite attributes in OPM cannot be nested; that is, a component attribute cannot be a composite attribute.

Depending on the type of the associated value class, an attribute can be primitive or abstract. A **primitive** attribute is an attribute associated with one of the following **primitive value classes**:

1. a **controlled value** class of enumerated atomic values, such as integers (e.g., CONTROLLED VALUE CLASS NUMBER_1{1, 2, 3}), or strings (e.g., CONTROLLED VALUE CLASS PROJ_TYPE{"*overlap*", "*homologs*", "*single*", "*nonoverlap*"});

2. a class of atomic values of one of the following types: BOOLEAN, BINARY(n), CHAR(n), VARCHAR(n), INTEGER, SMALLINT, REAL, FLOAT, DATETIME, TIMESTAMP, MONEY, TEXT, IMAGE;

3. one of the system metaclasses.

An **abstract** attribute is an attribute whose associated value class is an OPM class or a union of OPM classes.

Attributes can be characterized by the following types of constraints: **single-valued** (the default) or **multi-valued**; and **can be null** (the default) or **not null.**

A.2 Object Classes

An object class is identified by a unique object class name, and can be described using a **class description**. Each object class is associated with one or several (member) attributes.

An attribute can be associated directly only with one object class. However, subclasses inherit all the attributes of their superclasses. A subset of the attributes associated with an object class is specified as the **identifier** for the objects in that class; object identifiers are used to distinguish among the objects (instances) of an object class.

There are two main types of object classes in OPM: base object classes and specialization (subset) object classes. **Specialization** is an abstraction mechanism that allows defining object classes consisting of subsets of objects of other (**generic**) object (super) classes. A **base** object class is an object class that is not specified as a specialization (subclass) of any other object class. A base object class must be associated with an object identifier. A specialization object class is not associated directly with an identifier, and *inherits* the attributes of all its (direct and transitive) object superclasses, including the identifier; these attributes are called its **inherited** attributes. The specialization object classes form directed acyclic graphs.

The following is an example of a base object class called PROJECT:

```
OBJECT CLASS PROJECT
        DESCRIPTION: Defines laboratory projects.
        ID: project_id
        ATTRIBUTE project_id: INTEGER          single-valued    not null
        ATTRIBUTE project_parent: PROJECT      multi-valued     can be null
        ATTRIBUTE sponsored: SPONSOR           multi-valued
        ATTRIBUTE project_team: PERSON         multi-valued     not null
```

A.3 Protocol Classes

Laboratory (and other) protocols are modeled by **protocol classes.** Like base object classes, protocol classes have class names, (optional) class descriptions, identifiers and are associated with (member) attributes.

A protocol may consist of several steps or subprotocols. Protocol modeling is characterized by the recursive specification of protocols in terms of component subprotocols, called **protocol expansion.** Protocol expansion allows specifying *alternative protocols, sequences of protocols*, and *optional protocols*; "or", ",", and "[ ]" are used to denote alternative, sequences of, and optional protocols, respectively, and parentheses are used for specifying complex protocol compositions. For example, if ***P*** is a protocol whose expansion is ***(A, B, [C]) or D*** then protocol ***P*** is defined as either (i) the sequence of protocols ***A*** followed by ***B*** and followed by optional protocol ***C***, or (alternative) (ii) protocol ***D*** alone.

In addition to regular attributes (e.g., representing various protocol parameters, such as time and temperature), a protocol class has in general attributes representing the *input* and *output* of the protocol. These input and output attributes can only be associated with protocol classes. Input and output attributes of protocol classes are defined immediately below.

A.4 Input and Output Attributes

Input and output attributes of a protocol class specify the input and output of this protocol, and the relationship between the protocol and its subprotocols and/or inter-protocol relationships (connections). Input and output attributes can be only simple attributes, and can be associated only with protocol classes.

If a protocol P_i is expanded into several sub-protocols, then the input and output attributes of P_i must be referenced in the input and output attribute definitions of its sub-protocols. Relationships between input and output attributes of sub-protocols and input and output attributes of higher level protocols are expressed in OPM using *input is-a . . .* and *output is-a . . .* statements.

If a protocol P_i is followed by protocol P_j, then the input of P_j will include some or all of the output of P_i. Input-output protocol connections are expressed in OPM using *input from . . . via . . .* statements. For example, suppose that a protocol for DNA packaging consists of three sub-protocols: PACKAGE, DIGEST and ADD. Part of the input of DIGEST comes from the output of PACKAGE. Therefore, protocol DIGEST is defined as:

```
PROTOCOL CLASS DIGEST
        DESCRIPTION: digest
        ID: digest_id
        ATTRIBUTE digest_id: INTEGER          single-valued     not null
        ATTRIBUTE enzyme: ENZYME              input
        ATTRIBUTE dna: PACKAGED_DNA           input
                      from PACKAGE via packaged_dna
        ATTRIBUTE linear_dna: LINEAR_STICKY_DNAoutput
```

As already mentioned above, input and output attributes specify how sub-protocols are connected. When an input or output attribute corresponds to an attribute of a higher-level (generic) protocol, this correspondence needs to be specified as shown in the following example:

```
PROTOCOL CLASS DNA_PACKAGING
        DESCRIPTION: packaging DNA for insertion
        ID: protocol_id
        EXPANSION: PACKAGE, DIGEST, ADD
        ATTRIBUTE protocol_id: INTEGER          single-valued    not null
        ATTRIBUTE dna_sample: DNA_SAMPLE        input
        ATTRIBUTE vector: VECTOR                input
        ATTRIBUTE enzyme: ENZYME                input
        ATTRIBUTE markers: MARKERS              input
        ATTRIBUTE repackaged_dna: REPACKAGED_DNA                 output

PROTOCOL CLASS PACKAGE
        ID: package_id
        ATTRIBUTE package_id: INTEGER           single-valued    not null
        ATTRIBUTE dna_sample: DNA_SAMPLE        input
                        isa DNA_PACKAGING.dna_sample
        ATTRIBUTE vector: VECTOR                input
                        isa DNA_PACKAGING.vector
        ATTRIBUTE packaged_dna: PACKAGED-DNA                     output

PROTOCOL CLASS DIGEST
        ID: digest_id
        ATTRIBUTE: INTEGER                      single-valued    not null
        ATTRIBUTE enzyme: ENZYME                input
                        isa DNA_PACKAGING.enzyme
        ATTRIBUTE dna: PACKAGED_DNA             input
                        from PACKAGE via packaged_dna
        ATTRIBUTE linear_dna: LINEAR_STICKY_DNA                  output
```

```
PROTOCOL CLASS ADD
        ID: add_id
        ATTRIBUTE add_id: INTEGER               single-valued     not null
        ATTRIBUTE markers: MARKERS              input
                        isa DNA_PACKAGING.markers
        ATTRIBUTE linear_dna: LINEAR_STICKY_DNA                   input
                        from DIGEST via linear_dna
        ATTRIBUTE repackaged_dna: REPACKAGED_DNA                  output
                        isa DNA_PACKAGING.repackaged_dna
```

A.5 Derived Attributes

Derived attributes are associated with an object or protocol class and are derived from other attributes using a derivation rule. There are seven types of derivation rules:

1. arithmetic expression involving other attributes;
2. aggregate functions involving other attributes;
3. attribute inversion;
4. attribute match;
5. attribute composition;
6. attribute subvalue;
7. attribute union.

A simple attribute can be associated with one of the seven types of the derivation rules listed above. A composite attribute can be associated only with attribute matching. However, composite attributes that are not specified using attribute matching can contain components that are specified using attribute inverse, attribute composition, attribute subvalue, attribute union, arithmetic expression or aggregate function derivation.

An **arithmetic** derivation rule for a derived attribute associated with object or protocol class O_i consists of operators (+, -, *, /), constants, and other numeric attributes of O_i. Attributes involved in an arithmetic expression must be single-valued, simple or component attributes that are not associated with derivation rules.

An **aggregate** function derivation rule for a derived attribute associated with object or protocol class O_i consists of aggregate functions **min**, **max**, **sum**, or **avg** applied on a numeric attribute of O_i, or aggregate function **count** applied on an attribute of O_i. Attributes involved in an aggregate function derivation rule must be simple or component attributes that are multi-valued and are not associated with derivation rules.

The following object class definition contains two examples of derived attributes involving aggregate function expressions:

```
OBJECT CLASS SPONSOR
        DESCRIPTION: sponsor of a project
        ID: sponsor_id
        ATTRIBUTE sponsor_id: INTEGER          single-valued    not null
        ATTRIBUTE sponsor_name: CHAR(80)
        ATTRIBUTE (account, project, amount): (ACCOUNT, PROJECT, MONEY) multi-valued
        ATTRIBUTE total_amount: MONEY
                       DERIVATION: sum of amount
        ATTRIBUTE no_of_projects: INTEGER
                       DERIVATION: count of project
```

We use below the following notation: if A denotes an attribute of object or protocol class O_i, and x denotes an object instance of O_i, then $A\ (x)$ denotes the set of A values for x.

An attribute A of object or protocol class O_i can be defined as the **inverse** of an attribute B of object or protocol class O_j iff

1. the value class associated with A, $V(A)$, is O_j and the value class associated with B, $V(B)$, is O_i;
2. if A is a simple attribute, then B can be either specified as inverse of A or it is not specified as a derived attribute;
3. if A is a component attribute, then B must be specified as inverse of attribute A.

If A is defined as the inverse of B, then for every object x of O_i, whenever object y of O_j belongs to $A\ (x)$, x belongs to $B(y)$.

An attribute A of object or protocol class O_i can be defined as the inverse of attributes $B_1,..., B_m$, where B_k is associated with class O_k and has value class $V(B_k), 1 \le k \le m$, iff

1. A is associated with a union of value classes $V(A_1), \ldots, V(A_n)$, so that $m = n$ and for every pair $V(A_k)$ and $V(B_k)$, $1 \le k \le m$, $V(A_k)$ is O_k and $V(B_k)$ is O_i;

2. $\boldsymbol{B}$ can be specified as inverse of A or it is not specified as a derived attribute.

If A is defined as the inverse of $\boldsymbol{B_1}$ or . . . or $\boldsymbol{B_m}$ then for every object x of O_i, whenever object y of O_k, $1 \leq k \leq m$, belongs to A (x), x belongs to $\boldsymbol{B_k}$ (y).

The following object class definitions contain examples of derived attributes defined using inversion:

```
OBJECT CLASS CHROMOSOME
        ID: chromosome_number
        ATTRIBUTE chromosome_number: INTEGER  single-valued
        ATTRIBUTE has_map: MAP                multi-valued
                       DERIVATION: inverse of MAP.has_chromosome
        ATTRIBUTE owner: PERSON               single-valued

OBJECT CLASS MAP
        ID: map_id
        ATTRIBUTE map_id: INTEGER             single-valued
        ATTRIBUTE has_chromosome: CHROMOSOME              multi-valued
                       DERIVATION: inverse of CHROMOSOME.has_map
        ATTRIBUTE owner: PERSON               single-valued

OBJECT CLASS PERSON
        ID: social_security_no
        ATTRIBUTE social_security_no: CHAR    single-valued     not null
        ATTRIBUTE owns: MAP or CHROMOSOME                       multi-valued
                       DERIVATION: inverse of (MAP.owner or CHROMOSOME.owner)
```

A **simple** attribute A of object or protocol class O_i can be defined as **matching** an attribute $\boldsymbol{B}$ of object or protocol class O_j on attribute, C, iff $(\boldsymbol{B}, C)$ is defined as a composite attribute of O_j, the value class of C includes O_i, and the value classes of A and $\boldsymbol{B}$ are identical.

If A is defined as matching $\boldsymbol{B}$ of O_j on C then for every object x of O_i:

1. if there exists an object y of O_j so that object x of O_i belongs to $C(y)$, then $A(x)$ and $B(y)$ are equal;

2. if there does not exist an object y of O_j so that object x of O_i belongs to $C(y)$, then $A(x)$ is empty; if A does not allow null values, then for every object x of O_i there must exist an object y of O_j so that x belongs to $C(y)$.

A **composite** attribute $A = (A_1, \dots, A_n)$ of object or protocol class O_i can be defined as **matching composite** attribute $(B_1, \dots, B_n)$ of O_j on attribute C, iff $(B_1, \dots, B_n, C)$ is defined as a composite attribute of O_j, the value class of C includes O_i, and the value classes of A_k and B_k, $1 \le k \le n$, are identical.

If $A = (A_1, \dots, A_n)$ is defined as matching $(B_1, \dots, B_n)$ of O_j on attribute C, then for every object x of O_i:

1. if there exists an object y of O_j so that object x of O_i belongs to $C(y)$, then the set of tuples $(A_1(x), \dots, A_n(x))$ and the set of tuples $(B_1(y), \dots, B_n(y))$ are equal;
2. if there does not exist an object y of O_j so that object x belongs to $C(y)$, then $A(x)$ is empty; if A does not allow null values, then for every object x of O_i there must exist an object y of O_j so that x belongs to $C(y)$.

The following object class definitions contain examples of derived attributes defined using match:

```
OBJECT CLASS TRANSLATES
        DESCRIPTION: gene translates protein at some cell
        ID: (gene, at_cell, protein)
        ATTRIBUTE (gene, at_cell, protein): (GENE, CELL, PROTEIN) single-valued   not null

OBJECT CLASS GENE
        ID: gene_name
        ATTRIBUTE gene_name: VARCHAR(80)      single-valued      not null
        ATTRIBUTE (translate, at_cell): (PROTEIN, CELL)
                        DERIVATION: match (protein, at_cell) of TRANSLATES on gene


OBJECT CLASS PROTEIN
        ID: protein_name
        ATTRIBUTE protein_name: VARCHAR(80)    single-valued      not null
        ATTRIBUTE (gene, at_cell): (GENE, CELL)
                        DERIVATION: match (gene, at_cell) of TRANSLATES on protein
```

```
OBJECT CLASS CELL
        ID: cell_name
        ATTRIBUTE cell_name: VARCHAR(80)          single-valued       not null
        ATTRIBUTE (gene, protein): (GENE, PROTEIN)
                              DERIVATION: match (gene, protein) of TRANSLATES on at_cell
```

An attribute of an object or protocol class O_i, A, can be derived by **composing** attributes A_1, A_2, .. . , A_n, $n \geq 2$, where each A_j $(1 \leq j \leq n)$

1.is a local attribute (not an inherited attribute);

2.is either a simple or a component attribute;

3.is either non-derived or an inverse attribute;

4.cannot be an input or output attribute of a protocol class.

The composition of attributes $A_1, A_2, \ldots, A_n$ is denoted $A_1 . A_2 . \ldots . A_n$.

Attribute A_1 must be a simple or composite attribute of O_i. If the value class of A_j $(1 \leq j \leq n-1)$ consists of class O_{A_j}, then A_{j+1} must be an attribute of O_{A_j}. If A_j is associated with a value class consisting of a union of classes, $O_{A_{j_1}}$ or ... or $O_{A_{j_m}}$, then classes $O_{A_{j_1}}$, ... , and $O_{A_{j_m}}$ must have an attribute A_{j+1} associated with the same value classes.

The following object class definition contains an example of a derived attribute defined using composition:

```
OBJECT CLASS PERSON
        ID: social_security_no
        ATTRIBUTE social_security_no: CHAR(11)    single-valued       not null
        ATTRIBUTE primary_account: ACCOUNT        single-valued
        ATTRIBUTE sponsor_names: CHAR(80)         multi-valued
                              DERIVATION: primary_account.sponsor.sponsor_name

OBJECT CLASS ACCOUNT
        ID: account_no
        ATTRIBUTE account_no: INTEGER             single-valued       not null
        ATTRIBUTE sponsor: SPONSOR                multi-valued
```

```
OBJECT CLASS SPONSOR
        ID: sponsor_name
        ATTRIBUTE sponsor_name: CHAR(80)        single-valued     not null
```

In the example above, the composition derivation for sponsor_names involves attribute primary_account which is an attribute of PERSON and has value class ACCOUNT; attribute sponsor which is an attribute of ACCOUNT and has value class SPONSOR; and attribute sponsor_name which is a primitive attribute of SPONSOR.

Let B be a simple or component attribute of an object or protocol class O_i, so that B is not a derived attribute, nor an input or output attribute. A simple or component attribute A of O_i can be defined as a **subvalue** attribute of B, if the value class of A is V_A, the value class of B is V_B, and

1. $O_A = O_B$, $V_B = O_B$, and O_A is an immediate or transitive subclass of O_B; or

2. V_B consists of a union of classes, O_{B_1} or ... or O_{B_k}, and V_A (consisting of a single class or a union of classes) is a subset of V_B.

The following is an example of an attribute defined using the subvalue derivation:

```
OBJECT CLASS DEPARTMENT
        ID: department_name
        ATTRIBUTE department_name: CHAR(20)     single-valued     not null
        ATTRIBUTE employees: EMPLOYEE           multi-valued      not null
        ATTRIBUTE engineers: ENGINEER           multi-valued
                          DERIVATION: subvalue of employees
```

Class ENGINEER is a subclass of EMPLOYEE. The range of attribute employees consists of the employees in a given department, and the range of attribute engineers consists of the subset employees who are engineers in the same department.

A simple or component attribute A of an object or a protocol class O_i can be defined as the **union** attribute of other attributes, $B_1, \ldots, B_n$ $(n \geq 2)$ if

1.B_1, ... , and B_n are simple or component attributes of O_i;

2.B_1, ... , and B_n are not derived, nor input or output attributes of a protocol;

3.the union of the value classes of $\boldsymbol{B_1}, \ldots, \boldsymbol{B_n}$ is equal to the value class of $\boldsymbol{A}$.

The following is an example of an attribute specified using union derivation:

```
OBJECT CLASS PROJECT
        ID: project_id
        ATTRIBUTE project_id: INTEGER              single-valued    not null
        ATTRIBUTE company_sponsors: COMPANY                         multi-valued
        ATTRIBUTE government_sponsors: GOV_DEPARTMENT               multi-valued
        ATTRIBUTE all_sponsors: COMPANY or GOV_DEPARTMENT  multi-valued
                       DERIVATION: company_sponsors or government_sponsors
```