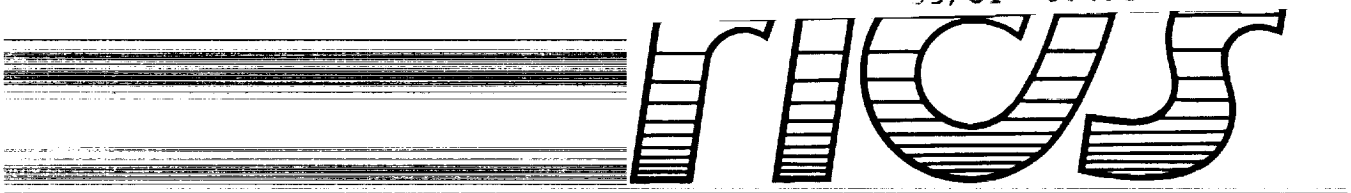
Advanced Software Development Workstation Project ACCESS User's Guide

Inference Corporation

October, 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.25

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Advanced Software Development Workstation Project
ACCESS User's Guide

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Inference Corporation. Dr. Charles McKay served as RICIS research coordinator.

Funding has been provided by the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Robert T. Savely, of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the author and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

Advanced Software Development Workstation Project ACCESS User's Guide

UHCL RICIS Contract NCC-9-16 SE.25

Prepared for
Research Institute for Computing and Information Systems
University of Houston - Clear Lake
and
Software Technology Branch
Information Technology Division
Information Systems Directorate
Johnson Space Center
National Aeronautics and Space Administration

October, 1990

Submitted by
Inference Corporation
550 N. Continental Blvd.
El Segundo, CA 90245
(213) 322-0200

Table of Contents

List of Figures

1. Overview

ACCESS is a knowledge-based software information system designed to assist the user in modifying retrieved software to satisfy user specifications. The purpose of This document is to provide a user's guide for the knowledge engineer who wishes to create for ACCESS a knowledge base consisting of representations of objects in some software system. The purpose of this activity is to make this knowledge base accessible to an end user who wishes to use the catalogued software objects to create a new application program or an input stream for an existing system.

The application-specific portion of an ACCESS knowledge base consists of a taxonomy of object classes (see Subsection 2.1.1), as well as instances (also called cases) of these classes. Within the knowledge base there are also "rule" objects which are used to dynamically modify user-created objects - e.g., by computing a new attribute value for an object from the values of other object attributes (Section 2.3). Chapter 2 describes how to create such a knowledge base.

All objects in the knowledge base are stored in an associative memory. When the end user selects an existing object for examination or creates a new one, the attributes of this object are matched against attributes of the objects in the associative memory using a redundant hash-addressing algorithm. Names of objects matching the current objects are displayed to the user, and can be selected by him for examination or copy and reformulation.

ACCESS provides a standard interface for the end user to browse and modify objects. This is described in Chapter 3. In addition, the interface can be customized by the addition of application-specific data entry forms and by specification of display order for the taxonomy and object attributes. These customization options are described in Chapter 4.

2. Specification of a Knowledge Base

The user interface for ACCESS is designed so that the end user should not need to know anything about the ART-IM® schema system. However, each object in ACCESS is represented as an ART-IM schema and the knowledge engineer must have enough knowledge of the ART-IM schema system to create the initial object taxonomy. A detailed description of the ART-IM schema system is available in the **ART-IM Reference Manual**, Volume I. This Chapter contains examples of how to use the ART-IM schema system to represent knowledge base objects.

Included in the ACCESS release directory is a file "kernel.art" which contains those schema definitions which are required for every ACCESS application. The knowledge engineer will extend these schema definitions to include an application-specific object hierarchy.

2.1 Specification of Objects and Object Attributes

2.1.1 Creation of an Object Taxonomy

Each object in the knowledge base has "attributes" which may or may not have "values". In the schema system, the attributes of an object are slots in the corresponding schema; the values of these attributes are the ART-IM objects which are slot values in the schema. A "feature" of an ACCESS object is a attribute/value pair (or slot/value pair).

The object taxonomy in ACCESS includes a single root schema, named **object**. The definition for **object**, as supplied in "kernel.art" is as follows:

```
(DEFSCHEMA OBJECT
        "The class of all objects."
        (NAME "OBJECT")
        (TEXT "")
        (SAVED-STATUS)
        (VIOLATES_CONSTRAINT))
```

Note that four attributes have been specified for this root object. These four attributes are inherited by child objects, although any default values for the attributes may be overridden. The value of the **name** attribute is the name of the object. The **text** slot has a string as its value (default is the empty string). The use of this slot is primarily confined to object instances and will be described in more detail below. The **saved-status** slot is used for internal bookkeeping. The **violates_constraint** slot is assigned a value each time the system detects that an object has an attribute with a value which violates a restriction the knowledge engineer has specified (e.g., the value of

a slot is not an integer). It is also assigned a value each time the system detects that a combination of object features violates a constraint the knowledge engineer has specified. The specification of restrictions on attribute values and the use of constraints on objects will be described in Sections 2.2 and 2.3 respectively.

For any specific application the knowledge engineer will define schemas which are related to **object** through explicit or inherited **is-a** and **instance-of** relationships. The examples in this section are derived from a knowledge base designed to represent a trajectory simulation software package and to generate an input stream for that package. In this case **object** has two application-specific children, **application_component** and **application_name**, defined as follows:

```
(DEFSCHEMA APPLICATION_COMPONENT
        (IS-A OBJECT)
        (NAME "Application_component"))

(DEFSCHEMA APPLICATION_NAME
        (IS-A OBJECT)
        (NAME "Application_name"))
```

Each of these objects in turn has children. For example, one of the children of **application_component** is **propagation-selection**, which is defined as follows:

```
(DEFSCHEMA PROPAGATION-SELECTION
        (IS-A APPLICATION_COMPONENT)
        (JCOAST)
        (JCOSTG)
        (ON-ORBIT-TARGETING)
        (VECTOR-PROPAGATION)
        (TYPE-OF-SIMULATION))
```

The slots in this schema represent attributes whose values are used to specify to the simulation program how to propagate the trajectory orbit.

This schema, or object, in turn has children. An example of a child object is **prop_sel_001**, defined as follows:

```
(DEFSCHEMA PROP_SEL_001
        (INSTANCE-OF PROPAGATION-SELECTION)
        (JCOAST 1)
        (JCOSTG 2)
        (ON-ORBIT-TARGETING 1)
        (VECTOR-PROPAGATION 0)
        (TYPE-OF-SIMULATION 3_DOF_TRANSLATIONAL_R-K)
        (NAME "PROP_SEL_003"))
```

The class **propagation-selection** is a subclass of the object **application_component**. An object in the **propagation-selection** subclass is intended to specify the input parameters required by the trajectory simulation package for propagating an orbit. In turn, the object **prop_sel_001** is a specific instance of

this generic subclass and represents a particular set of parameters governing trajectory propagation. For example, the value of the slot, **type-of-simulation** is a symbol **3_dof_translational_r-k** indicating that the integration technique to be used will be a translational Runge-Kutta method with 3 degrees of freedom.

Whatever his application, the knowledge engineer must specify a taxonomy of objects and specify the attributes of each object in this taxonomy. The specification of object attributes may be either explicit or implicit through inheritance from ancestor objects.

2.1.2 Attributes with Multiple Values

In ACCESS an object attribute can have a single value or multiple values. An attribute which is allowed to have only a single value is represented in the underlying ART-IM representation by a slot whose cardinality is **single.** This is the default and the knowledge engineer can define such attributes implicitly by simply using the slot in a schema. An attribute which may have more than one value is represented by a multi-valued slot. Such slots must be explicitly defined **before** they are used in any schema. An example of a multi-valued slot is the slot **violates_constraint**, which is specified as follows:

```
(DEFSCHEMA VIOLATES_CONSTRAINT
        (INSTANCE-OF SLOT)
        (CARDINALITY MULTIPLE)
        (INHERITS YES))
```

In this case, the schema definition specifies that **violates_constraint** is a multi-valued slot and that its uses will be inherited through inheritance links to related schemas. This is the default behavior; to create a slot whose uses are *not* inherited, the value of **inherits** must be **no.**

When the knowledge engineer creates a file defining the knowledge base, any multi-valued slot must be defined before any schema(s) which uses this slot.

2.2 Specification of Restrictions on Attribute Values

It is quite common that the values of an particular attribute for an object need to be restricted in some way. For example, the value of the **hours** attribute in a **date** object should be an integer between 0 and 23. In the example shown above, the value of the attribute **type-of-simulation** in a **propagation-selection** object represents the type of integrator which will be used to propagate an trajectory. This value must be one of an explicitly enumerated set of values.

ACCESS provides the knowledge engineer the capability to restrict the value of an object attribute in one of three ways:

1. Restrict the value to a specific type - **float**, **integer**, **symbol**, or **string**.

2. Restrict the value to be one of an explicitly enumerated set of values.

3. Restrict the value to be a symbol representing an object in one of an explicitly specified set of subclasses of objects.

In order to specify any of these types of restrictions, the knowledge engineer must create an instance of **attribute-restriction** schema. The **attribute-restriction** and **enumerated-set** schemas are defined as follows:

```
(DEFSCHEMA ATTRIBUTE-RESTRICTION
  (ALLOWABLE-CLASSES)
  (HAS-ENUMERATED-SET)
  (TYPE))

(DEFSCHEMA ENUMERATED-SET
  (ALLOWABLE-VALUES))
```

In order to be used by ACCESS, the name of an **attribute-restriction** schema instance must be the concatenation of the name of the attribute to which the restriction applies and "-restriction."

In specifying an **attribute-restriction** schema, exactly one of the three slots **type**, **has-enumerated-set**, and **allowable-classes** should be used. The **type** slot is used to restrict the value of the attribute in question to be a specific type. Allowable values for the **type** slot are **float**, **integer**, **symbol**, or **string**.

The **has-enumerated-set** slot is used to restrict the value of the attribute in question to a explicitly enumerated set of objects. The value of this slot must be a schema which is an **instance-of** an **enumerated-set**. The value of the **allowable-values** slot in this schema must be a sequence representing the list of allowable values.

The **allowable-classes** slot is used to restrict the value of the attribute in question to an instance of one of an explicit set of subclasses. For example, if the attribute **propagator_select** must have a value which is an instance of a **propagation-selection** object, this would be specified as follows:

```
(DEFSCHEMA PROPAGATOR_SELECT-RESTRICTION
        (INSTANCE-OF ATTRIBUTE-RESTRICTION)
        (ALLOWABLE-CLASSES PROPAGATION-SELECTION))
```

When the user assigns a value to an attribute which violates one of these restrictions, this will be detected by ACCESS. The result is that the object in question is marked as having a constraint violation. That is, a value is put into the **violates_constraint** slot of that object, indicating the type of restriction which has been violated. In this case, the value put into that slot would be **propagator_select-restriction-violation**.

In addition, through the interface, a pop-up window with a message to the user is displayed. This is described in Section 3.2.

2.3 Specification of Constraints on Objects

2.3.1 Rule Objects

An important aspect of the ACCESS design is that as objects are created and modified, there exist modification rules within the system which can propagate new attribute values or check the validity of existing values.

Modification rules in the knowledge base are themselves represented as objects and are transformed at initialization time into ART-IM rules. The definitions of the object class **rule** and the instance **compile_rule**, as supplied in **kernel.art** are as follows:

```
(DEFSCHEMA RULE
  "The class of all rules."
  (IS-A OBJECT)
  (COMPILE COMPILE_RULE_METHOD)
  (NAME "Rule"))

(DEFSCHEMA COMPILE_RULE
  (INSTANCE-OF RULE)
  (TEXT "(defrule compile_rule
  (declare (salience -100))
  (schema ?rule&~compile_rule
    (instance-of rule)
    (text ?))
  =>
  (send compile ?rule))"))
```

At initialization of ACCESS, the ART-IM rule defined in the **text** slot of **compile_rule** is compiled. This creates a production rule, itself called **compile_rule**, which will cause the value of the **text** slot for each object which is an instance-of a **rule** to be compiled. In this way rules are generated which can modify other objects in the knowledge base.

2.3.2 Constraints and Argument Lists

A subclass of the class **rule** is the class **constraint**. The definition of the **constraint** class is as follows:

```
(DEFSCHEMA CONSTRAINT
  "The class of all constraints."
  (IS-A RULE)
  (ATTRIBUTE)
  (CONSTRAINT_OF)
  (NAME "Constraint"))
```

The class **constraint** in turn has subclasses **formula**, **template**, and **predicate.**

A **formula** is an object used to specify how to compute the value of an attribute of a particular class of objects in terms of other attributes of that object and its subobjects. (A subobject of given object is an object which is the value of an attribute of the given object.) For example, a class of objects of type **phase** has attributes **day**, **hour**, **min**, and **sec**, whose values represent the days, hours, minutes, and seconds since launch for the beginning of this phase of the mission. It also has an attribute, **tevent**, whose value is the time of the beginning of launch expressed in seconds. This value can be calculated from the values of the **day**, **hour**, **min**, and **sec** slots.

A **template** is an object used to specify how a piece of text (normally a code fragment) is to be generated on the basis of the attribute values of an instance of a class of objects.

A **predicate** is used to specify how the system will verify relationships between the values of attributes of an object and its subobjects. A function is specified whose arguments are these values and which returns T if the reelationship constraints are satisfied and **NIL** if they are not. If the test fails, the object is marked as having a constraint violation..

Any **formula**, **template**, or **predicate** schema must specify a value for the **constraint_of** slot. This value is the name of the object class to which the constraint is intended to apply. The computation specified by one of these constraints will only be invoked for objects which are instances of the specified class.

Formulas and **predicates** rely on user-specified functions either (in the case of a **formula**) to compute an attribute value or (in the case of a **predicate**) to test for a constraint violation. **Templates** use the ART-IM function **sprintf** to generate a string value. The knowledge engineer must specify how to determine the arguments to these functions.

This is done through the value of the **arguments** slot. This value must be a sequence of **n** elements, where **n** is the number of arguments which will be required by user-specified function or by **sprintf**. Each element in the sequence must be either a symbol or a sequence of symbols, each of which is an attribute name (in ART-IM, either a **slot** or a **relation**).

The use of this specification is perhaps best illustrated by an example. Suppose a constraint is intended to apply to all objects in the class **rendezvous**, and that this class has attributes **propagator_select** and **omp_model_select.** The values of these attributes are intended to be instances of **propagation-selection** and **omp_model** schemas, each of which will have a **text** attribute, whose values are

intended to be used as arguments to the constraint function. The following code fragment shows how this would be specified:

```
(constraint_of rendezvous)
(arguments ( (propagator_select text) (omp_model_select text)))
```

A constraint function will be applied to a specific object only if all arguments for that function exist. In general, if the *i-th* element of the argument list is a symbol representing a slot, then the *i-th* argument for the constraint function will the the value of the slot in the object in question. If the *i-th* value of the argument list is a sequence of symbols, **a1, a2,...**, then the argument is found by first taking the subobject which is the value of the slot **a1**, then taking the value of the slot **a2** in that subobject, assuming it also exists, etc. If the entire chain of subobjects exists, then the final element in the chain is the value passed to the constraint function.

The number of arguments to a formula or predicate function or to `sprintf` is limited to 40.

2.3.3 Formulas

A **formula** schema results in the creation of a rule which is used to compute a value for some attribute of a particular object class. A **formula** schema is of the following form:

```
(defschema formula
        (constraint_of) ;class to which formula applies
        (arguments)
        (attribute)     ;attribute for which value is computed
        (function)      ;name of function used to produce attribute value
        (name)          ;for documentation
)
```

The value of the **function** slot of a formula instance is symbol representing the name of a user-supplied function. This is normally a **def-art-fun**, but could be a **def-user-fun** or an ART-IM system function.

In order for a formula to be applied, there must be an object in the knowledge base which is an instance of the class given by the value of the **constraint_of** slot. The values of attributes of this object and its subobjects as specified by the value of the **arguments** slot must exist. These values are then passed as arguments to the function specified in the **function** slot. The return value from this function is then used to modify the value of the object attribute specified by the value of the **attribute** slot.

As an example, consider a knowledge base which contains an object **date**, defined as follows:

```
(DEFSCHEMA PHASE
  (IS-A APPLICATION_COMPONENT)
  (PH_NUMBER)
  (PH_TITLE)
  (ICOAST)
  (COVAR_MAT)
  (DAY)
  (HOUR)
  (MIN)
  (SEC)
  (T_HOUR)
  (T_MIN)
  (T_SEC)
  (TEVENT)
  (TERMIN)
  (BTIME)
  (STRING1 "         * ------------------------------------------------------
           *
           * ------------------------------------------------------
        $PHASE")
  (NAME "Phase"))
```

The following is a formula designed to compute the value of the slot **tevent** given values for the **day**, **hour**, **min**, and **sec** slots:

```
(DEFSCHEMA PHASE_FORMULA
  (INSTANCE-OF FORMULA)
  (ARGUMENTS(DAY HOUR MIN SEC ))
  (ATTRIBUTE TEVENT)
  (CONSTRAINT_OF PHASE)
  (FUNCTION CALC-SEC))

(def-art-fun calc-sec (?day ?hr ?min ?sec )
 (+ (+ (+ (* (* ?day 24) 3600) (* ?hr 3600)) (* ?min 60)) ?sec))
```

2.3.4 Templates

A **template** schema results in the generation of a rule which is used to a compute string value for a slot in a particular object class. A **template** schema is of the following form:

```
(defschema template
        (constraint_of) ;class to which template applies
        (attribute)     ;attribute for which value is computed
        (arguments)
        (name)          ;for documentation
        (template_string);format string for printf function
)
```

When an instance of a **template** schema is specified, the value of the **template-string** slot must be a string which is a suitable format string for **sprintf**. Continuing the example started in Subsection 2.3.2, here is a full specification of a simple template:

```
(defschema rendezvous_template
        (instance-of template)
        (constraint_of rendezvous)
        (arguments ( (propagator_select text) (omp_model_select text)))
        (template_string  "    * ---------------------------------------
        *   RENDEZVOUS
        * ---------------------------------------
        *
        %a
        %a
        "))
```

This **template** will apply whenever there is an instance of a **rendezvous** object such that the values of its **propagator_select** and **omp_model_select** attributes are themselves objects with **text** attributes. The values of these **text** attributes will be passed as arguments to **sprintf**, which will create a formatted string as specified by the value of the **template_string** slot. This string will then be asserted as the value of the **text** slot in the **rendezvous** instance object.

2.3.5 Predicates

A **predicate** schema results in the generation of a rule which is used to perform a procedural test on an object. If the test fails, the object is marked as having a constraint violation. This is done by asserting the name of the function as a value in the **violates_constraint** slot of the object.

A **predicate** schema is of the following form:

```
(defschema predicate
        (constraint_of) ;class to which predicate applies
        (arguments)
        (boolean-function);name of function to be used for test
        (name)          ;for documentation
)
```

As with a formula, the value of the **boolean-function** slot of a formula instance is symbol representing the name of a user-supplied function. This is normally a **def-art-fun**, but could be a **def-user-fun** or an ART-IM system function. A constraint violation is considered to have occurred if the function returns NIL.

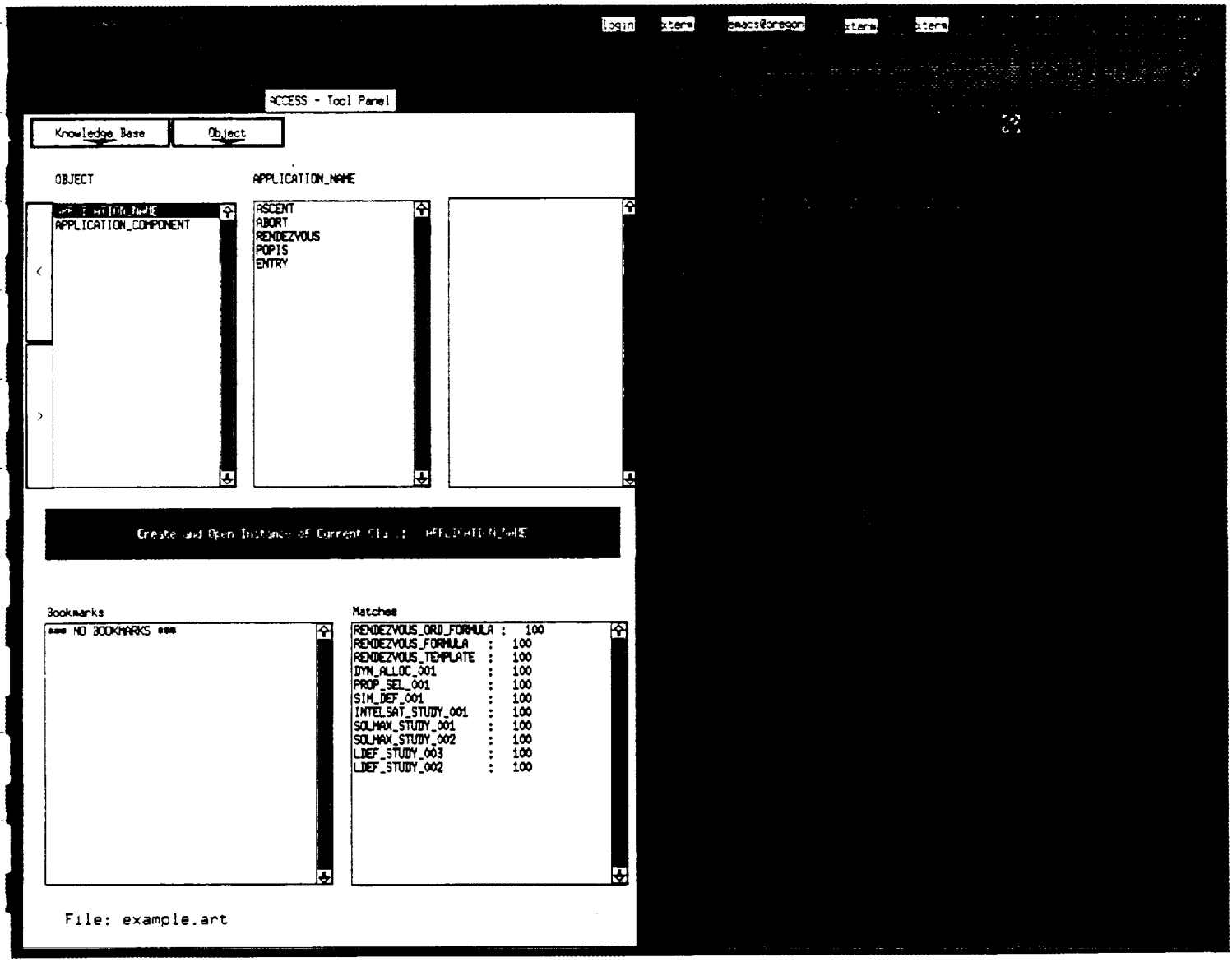
3. ACCESS Interface

The following sections briefly describe the nature of the ACCESS interface. Information on how to customize this interface appears in Chapter 4.

3.1 ACCESS Tools Panel

Figure 3-1: ACCESS Tools Panel

ORIGINAL PAGE IS
OF POOR QUALITY



The ACCESS Tools Panel displayed in Figure 3-1 is the first panel displayed upon invocation of ACCESS. The top half of this panel is used to display the taxonomy of the knowledge base. Within this region are three taxonomy subpanels or "windows." If an object in one of these subpanels is selected (by pointing and clicking) with the mouse, it is highlighted and becomes the "current object." If this object has children, then a list of these children is displayed in the next window to the right, with the name of the previously selected object displayed above. If an object is selected from the rightmost taxonomy window, then the taxonomy display is shifted one panel left before displaying the child list. Two buttons on the left of the taxonomy windows offer the user the option of shifting the display of the currently displayed taxonomy either right or left.

In the center there is a subpanel or "button" called the Open Object Button. This subpanel displays the name of the current object - that object which has been most recently selected from the taxonomy windows. If the current object is an object instance, then this button offers the option of "opening" the object for browsing or editing. Clicking on this button will invoke either the generic Form Panel or a custom form through which the object can be modified. If the current object is a class, then this button offers the option of creating an instance of that class and then opening the resulting instance for browsing or modification.

In the lower half of the Tools Panel are two windows, the Bookmarks Window and the Matches Window. The Bookmarks Window displays a chronologically ordered list of objects which have been opened during this ACCESS session. The Matches windows provides a list of objects in the same class as the current object, ordered by the extent to which they "match" the current object. Matching between two objects is done by comparing the features of one object with those of the other - each feature which matches increases the level of matching.

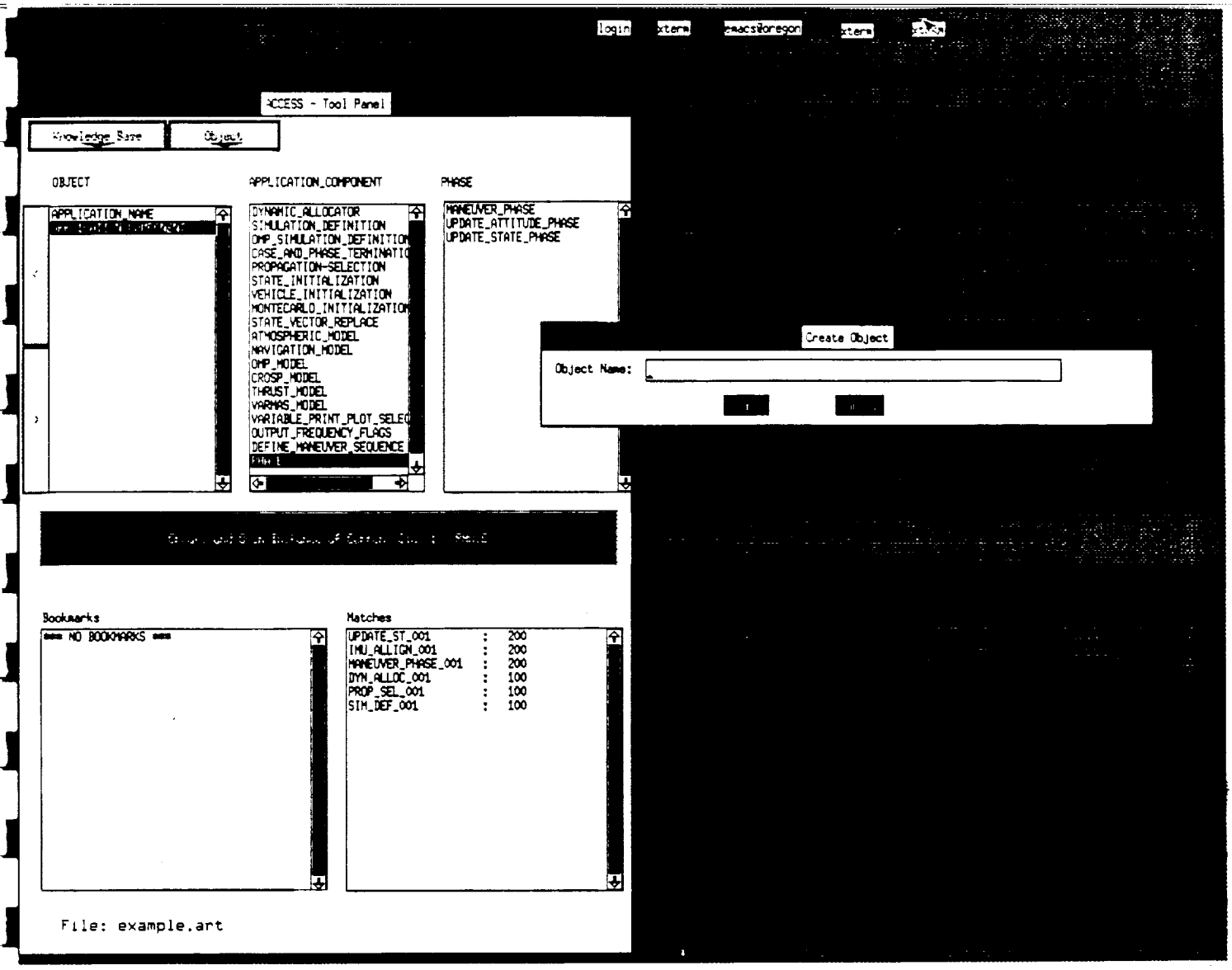
When an object is selected from either the Bookmarks Window or Matches Window by a mouse click, it becomes the current object. When this happens, the taxonomy windows are updated to display the ancestors and siblings of this object. The Open Object Button is updated to show the name of the new current object, and the Matches Window is also updated.

At the bottom of the Tools Panel is a display showing the name of the file from which the current knowledge base was loaded or to which it has been saved in the course of the current session.

3.2 Browsing or Modifying an Object - the Form Panel

In order to open an object or an instance of a class for browsing and/or editing, the user clicks on the Open Object Button in the center of the Tools Panel. If the current

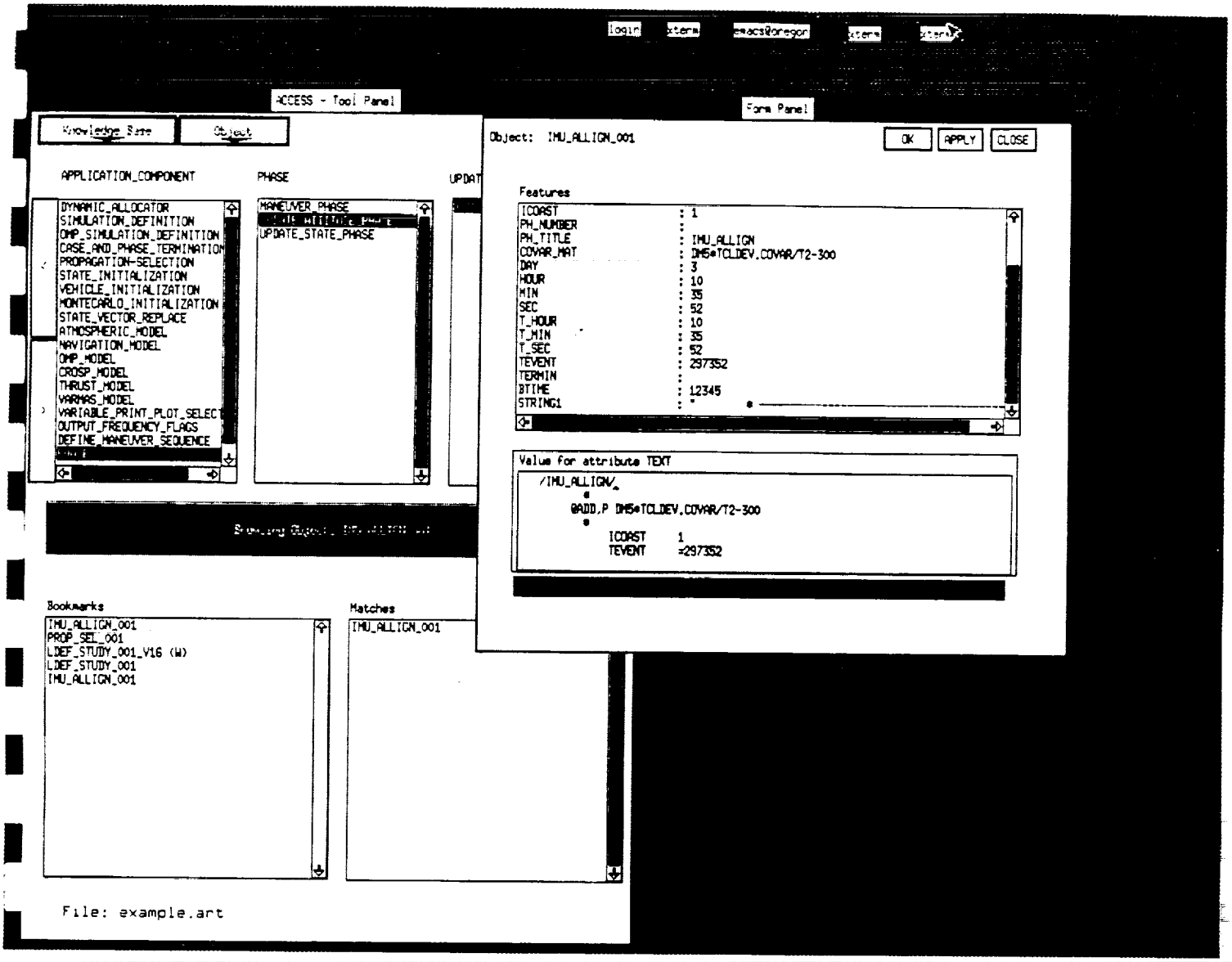
Figure 3-2: Prompt for Object Name



object represents a class, rather than an object instance, the user will be prompted for the name of a new instance, as illustrated in Figure 3-2. If the user supplies such a name, that becomes the name of the new current object.

Once an object instance is opened, the generic Form Panel will be displayed (unless a custom form has been specified for the class to which the object belongs). When the Form Panel appears, the Tools Panel is "frozen" - that is, it becomes insensitive to mouse clicks and other input. The generic Form Panel is shown in Figure 3-3.

Figure 3-3: ACCESS Generic Form Panel



The name of the object being browsed is displayed in the upper left hand corner of the Form Panel. In the top half of the panel is the Object Features subpanel or window. Displayed inside this window are a list of object attributes and values. Attribute names and values are truncated if necessary to conform with the screen size. The user can select a particular feature to examine by pointing and clicking with the mouse.

When a feature is selected, the value corresponding to that feature is displayed in the

Attribute Value subpanel, which appears in the lower half of the Form Panel. The user can enter text directly into this subpanel, thus editing the currently selected attribute value. Any such editing must be confirmed by hitting the ESC key. When this is done, the Features display will be updated to show the modified value.

Alternatively, if the currently selected attribute is one whose value is restricted to an enumerated set or to an instance of an allowable class of objects, a "SELECT" button will appear to the top and right of this subpanel. By clicking on the SELECT button, the user will cause a menu of allowable values for this attribute to be displayed; he can then select one of them from the menu. If a selection is made, the Features display will be modified to show the newly-selected value.

Editing within the Attribute window does not change values in objects in the knowledge base until the user clicks on either the "APPLY" or "OK" button in the upper right hand corner of the Form Panel. Clicking on the APPLY button causes the changes which have been recorded on the Form Panel to be made to the object in the knowledge base or, if the object being edited is a "SAVED" object, to a copy of that object.

Within the knowledge base, objects are considered to be either "SAVED" or "WORKING." SAVED objects are those which were read into the knowledge base at initialization time or which have been explicitly saved by the user (see Subsection 3.3.1). No modifications can be made to a SAVED object. A WORKING object, on the other hand, is one which has been created by the user in the course of the current session and has not been explicitly saved.

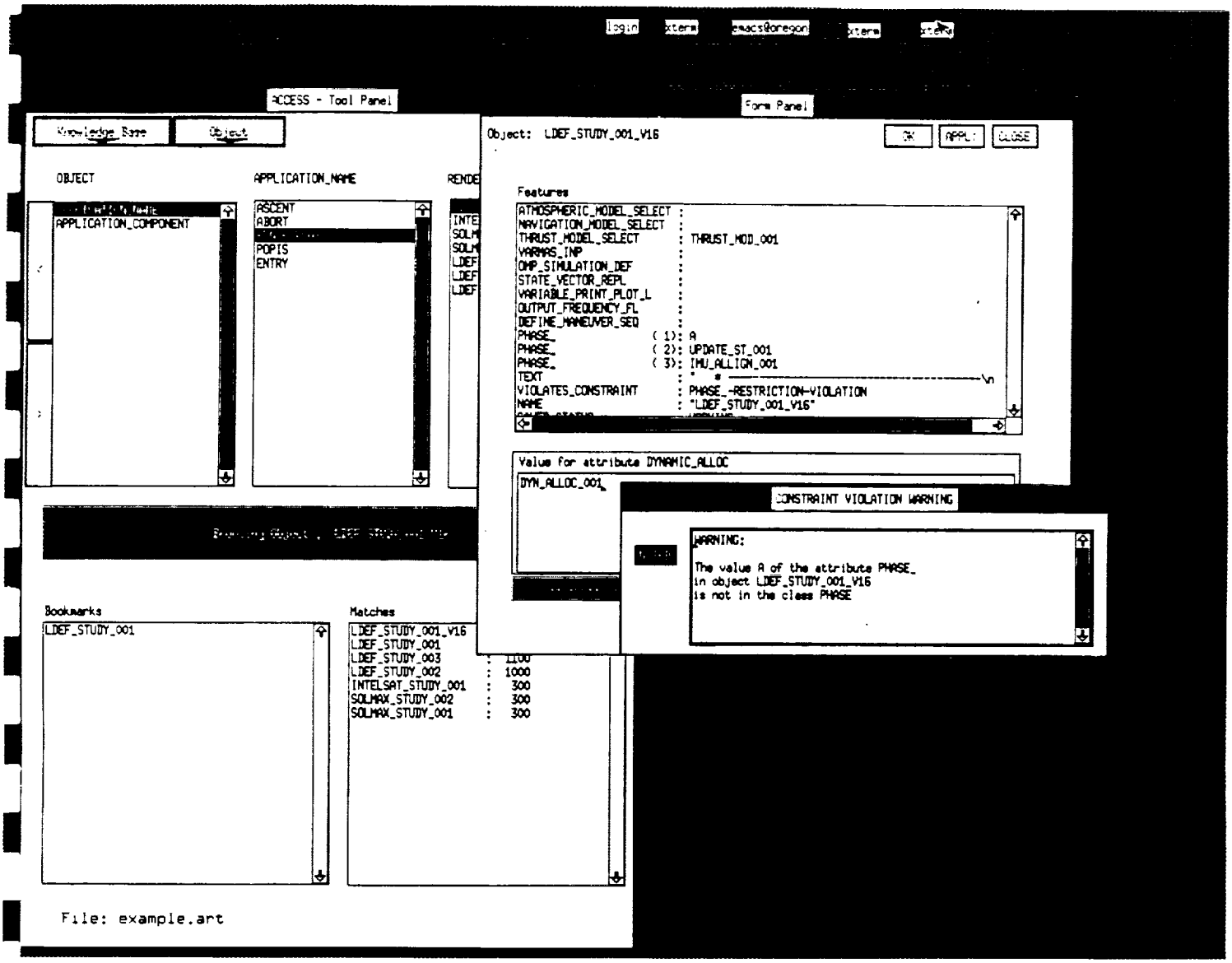
If the user is editing a SAVED object when he selects APPLY or OK, he will be prompted for a new object name. ACCESS will make a copy of the saved object, assign it the new name, and apply the changes to it.

Once any changes to attribute values have been made, any constraints based on these new values are propagated. If constraint violations are detected, then a pop-up panel with a warning message is displayed, with one warning message for each constraint violation. (Figure 3-4 shows the pop-up warning panel.) The Matches Window is also updated based on the new attribute values.

The final button in the top right hand corner of the Form Panel is the "CLOSE" button. Clicking on the CLOSE button causes the Form Panel to be erased from the screen and resensitizes the Tools Panel.

OK is equivalent to APPLY followed by CLOSE.

Figure 3-4: Panel with Warning of Constraint Violation



3.3 Tools Panel Menus

Near the top of the Tools Panel are two pulldown menus - the Object menu and the Knowledge Base Menu. These menus are described in the following subsections.

3.3.1 The Object Menu - Saving, Deleting, or Displaying Source Code

The Object Menu consists of three options - **save**, **delete**, **compare**, and **view source**. Each performs its function on the current object, that is, the object whose name is displayed on the Open Object button.

The **save** option makes the current object a SAVED object. This means that this object can no longer be modified by editing and that its description will be saved if one of the **save** options is selected from the Knowledge Base Menu.

The **delete** option deletes the current object from the knowledge base. If this object appeared on the Bookmarks list, it is deleted from that list. A new object is selected to be the current object and the deleted object will no longer appear in the object taxonomy display.

The **compare** option allows the user to compare the current object with another object in the same class. When this option is selected, a pop-up menu of objects with the same parents is displayed. If the user selects one of these objects, then a panel appears which gives a static display of all features (attribute/value pairs) of the two objects which are different.

The **view source** option displays value of the **text** slot of the current object on the Source Panel. The displayed text can be browsed, but not modified by the user. This option is appropriate when ACCESS is being used to generate source code or an input stream for some software system, as it allows the user to examine the generated code. The Source Panel is shown in Figure 3-5.

At the top of the Source Panel are "WRITE" and "CANCEL" buttons. By selecting the WRITE button, the user causes the text in the **text** slot of the current object (i.e., the text which is displayed on the Source Panel) to be written to a file. The base name for this file is the name of the current object; its suffix is "txt".

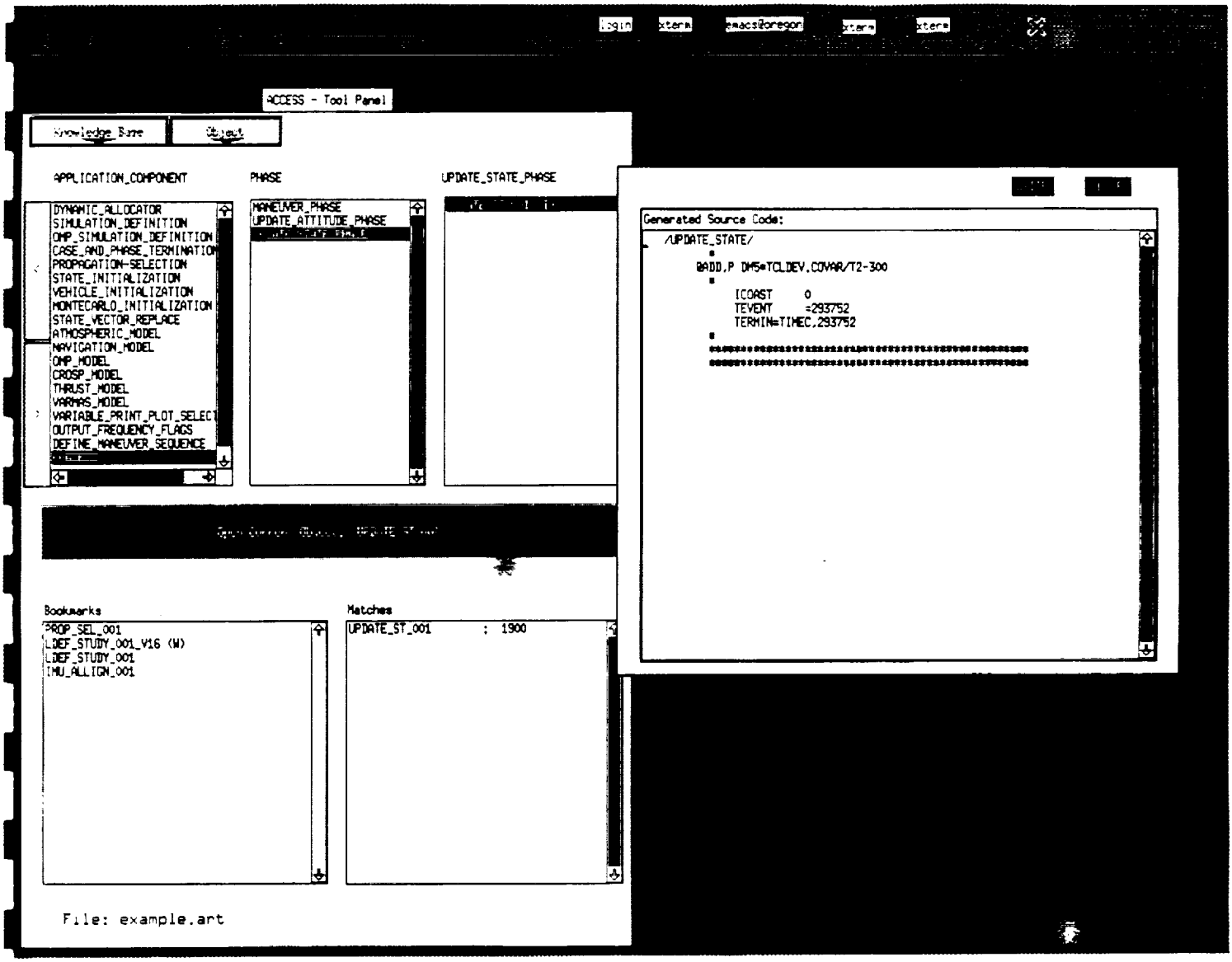
Selecting the CANCEL button returns control to the Tools Panel.

3.3.2 The File Menu - Saving the Knowledge Base

The Knowledge Base Menu consists of three options - **save**, **save as..**, and **exit**.

When the option **save** is selected, all SAVED objects in the knowledge base as well as ancillary customization data will be written out to the "current" file. This is the file whose name is displayed at the bottom of the Tools Panel - initially, it is the file from which the knowledge base was loaded. If there are WORKING objects in the knowledge base, a warning panel will be displayed and the user will have the option of canceling the save. The file created by **save** can be used as input to a subsequent ACCESS session.

Figure 3-5: Display of Source Code via Source Panel



The option **save as..** works in the same way as **save**, except that the user is prompted for the name of a file to save to. When supplied, this becomes the current file.

The **exit** option causes the current ACCESS session to be terminated. If a working object has been created or modify since the knowledge base was initially loaded or last saved, then a warning message is displayed and the user has the option of canceling.

4. Customization of the ACCESS Interface

4.1 Changing Defaults in the Interface

The default display order of the object taxonomy on the Tools Panel and the display order for an object's attributes and values on the Form Panel is governed by the way symbols are hashed internal to ART-IM. However, for clarity, the knowledge engineer may wish to specify a particular display order for attributes or a specific structure for the display of the object hierarchy. The method for doing so is described in the following sections.

4.1.1 Controlling the Display of the Object Taxonomy

The knowledge engineer has two methods for controlling the display of the object taxonomy in ACCESS. Each of these methods involves specifying a non-default value for a slot in the ACCESS schema.

4.1.1.1 Use of the ACCESS TAXONOMY Attribute

The ACCESS schema, which is present in all ACCESS knowledge bases, contains a **taxonomy** slot. This slot enables the knowledge engineer to specify a specific ordering for the displays of object classes which appear in the various taxonomy windows. The value of the **taxonomy** slot must be a sequence, with one element for each root object. Currently, there is single root object, called **object.** Each element of the taxonomy specification sequence is itself a sequence, the first element of which is an object (the parent) and the second element of which is a sequence with one element for each child of the given parent. The elements of this second sequence are **taxonomy** specifications in which the child is treated as the root object. The following shows a textual display of an object hierarchy and a code fragment with its corresponding sequence representation.

```
OBJECT
        APPLICATION_NAME
                ASCENT
                ABORT
                RENDEZVOUS
                POPIS
                ENTRY
        APPLICATION_COMPONENT
                DYNAMIC-ALLOCATOR
                SIMULATION_DEFINITION
```

```
(TAXONOMY
        ((object
                ((application_name
                                ((ascent ())
                                (abort  ())
                                (rendezvous())
                                (popis  ())
                                (entry  ())
                                )
                )
                (application_component
                                ((dynamic_allocator ())
                                (simulation_definition())
                                (omp_simulation_definition())
                                )
                )
                )
        ))
)
```

When the knowledge base is read into ACCESS at initialization time, ACCESS checks the validity of any non-default taxonomy which has been specified. This verification includes checking the syntactic correctness of the sequence structure, checking that each atom in this structure is a symbol which is the name of an object, and checking that the appropriate parent/child relationships hold. If this verification fails for any reason, then ACCESS will revert to the default display.

4.1.1.2 Use of the filtered_object Slot

There are some objects which should not be accessible at all to the end user of an ACCESS application. Examples include **rule** objects. To avoid the display of an object in the object taxonomy, the name of that object should be placed as a value in the **filtered_object** slot of the ACCESS schema.

4.1.1.3 Construction of the Taxonomy Display

When ACCESS is initially invoked, the leftmost taxonomy window shows those objects which are children of the root object, **object**. This child list is ordered to first display any children which are instances of **object** (i.e., objects which have an explicit **instance-of** relationship to **object**). Next in the ordering are those subclasses of **object** as specified by the taxonomy ordering. Finally appear children which are not filtered objects but which do not appear in the taxonomy ordering.

When the user selects an object from this list, it becomes the "current object." Its name is displayed above the middle taxonomy window. Its children are then displayed as described above - first, child instances, then child subclasses from the taxonomy ordering, finally child subclasses which are not specified in the taxonomy ordering.

4.1.2 Controlling the Display of Object Features

4.1.2.1 Controlling the Order Display of Object Attributes

When the knowledge engineer wishes to specify a particular order for displaying the features (attribute/value pairs) of an object on the Form Panel, the method for doing so is to use an **instance-of** an **ordering** schema. The definition of the **ordering** schema is as follows:

```
(defschema ordering
        (attribute-order))
```

In order to specify the display ordering of attributes for a particular class of objects, one must create an **instance-of** an **ordering** schema whose name is the name of the class followed by "-ordering." The value of the **attribute-order** slot in this schema should then be a sequence of names of attributes (slots) in the corresponding class. For example, suppose the knowledge engineer has defined a class, **date**, as follows:

```
(defschema date
        (is-a time_coordinates)
        (milliseconds)
        (seconds)
        (minutes)
        (hours)
        (day)
        (month)
        (year))
```

To specify a particular ordering for display of objects which are instances of the class **date**, the knowledge engineer may then specify the following:

```
(defschema date-ordering
        (instance-of ordering)
        (attribute-order (year month day hours minutes seconds
                         milliseconds)))
```

When an object of type date is displayed, the features will be ordered with the **year** first, followed by **month**, **day**, etc. If this object has other attributes which are not filtered objects, i.e., are not values of the **filtered_object** slot of the ACCESS schema, then the corresponding features will appear after those whose order has been explicitly specified. Note that typically the attributes **is-a** and **instance-of** are included among the list of filtered objects.

4.1.2.2 Controlling the Order of Display of Values of Multi-Valued Attributes

Some object attributes may have more than one value (see Subsection 2.1.2). These attributes are represented in ART-IM by multi-valued slots. For such attributes, the option exists to specify a function which will be sued to order the values of this

attribute for display purposes and for formula functions, etc. The name of the function must be of the form SLOT-NAME-ORDERING-FUNCTION, where SLOT-NAME is the name of the slot in question. This function must accept as arguments two ART-IM objects which are permissible values for the slot and must return an ART-IM integer. This function will be used in the same way as the comparison function to the C library function qsort - that is, the first argument will be considered to be less than, equal to, or greater than the second depending on whether the value returned is less than, equal to, or greater than zero. The function must be consistent - that is, if it returns 0 for two arguments a and b, then it must return 0 for the argument b and a. Similarly, the value of the function with arguments a and b must be opposite in sign to the value of the function with arguments b and a. Violation of these consistency restrictions can cause ACCESS to die.

An example of an ordering function is the following, which orders instances of **phase** schemas on the basis of the value of the slot **tevent**. The ordering is defined so any object with no value or a non-integer value for **tevent** is considered to be less than an object which does have an integer value.

```
(def-art-fun phase_-ordering-function (?s1 ?s2)
;get times from phase schemas 1 and 2
        (bind ?valid1 (and (symbolp ?s1) (schemap ?s1) (slotp ?s1 tevent)))
        (bind ?valid2 (and (symbolp ?s2) (schemap ?s2) (slotp ?s2 tevent)))
        (bind ?t1 (if ?valid1 then (get-schema-value ?s1 tevent) else NIL))
        (bind ?t2 (if ?valid2 then (get-schema-value ?s2 tevent) else NIL))
        (if (not ?t1) then
                (if ?t2 then -1 else 0)
        else (if (not ?t2) then 1
                else (if (< ?t1 ?t2) then -1
                        else (if (eq ?t1 ?t2) then 0
                                        else 1)))))
```

Internally, ACCESS uses a function which accepts as arguments a sequence of objects and the name of an ordering function for those objects and returns a sequence of the same objects in ascending order. There is a public interface to this function of the form

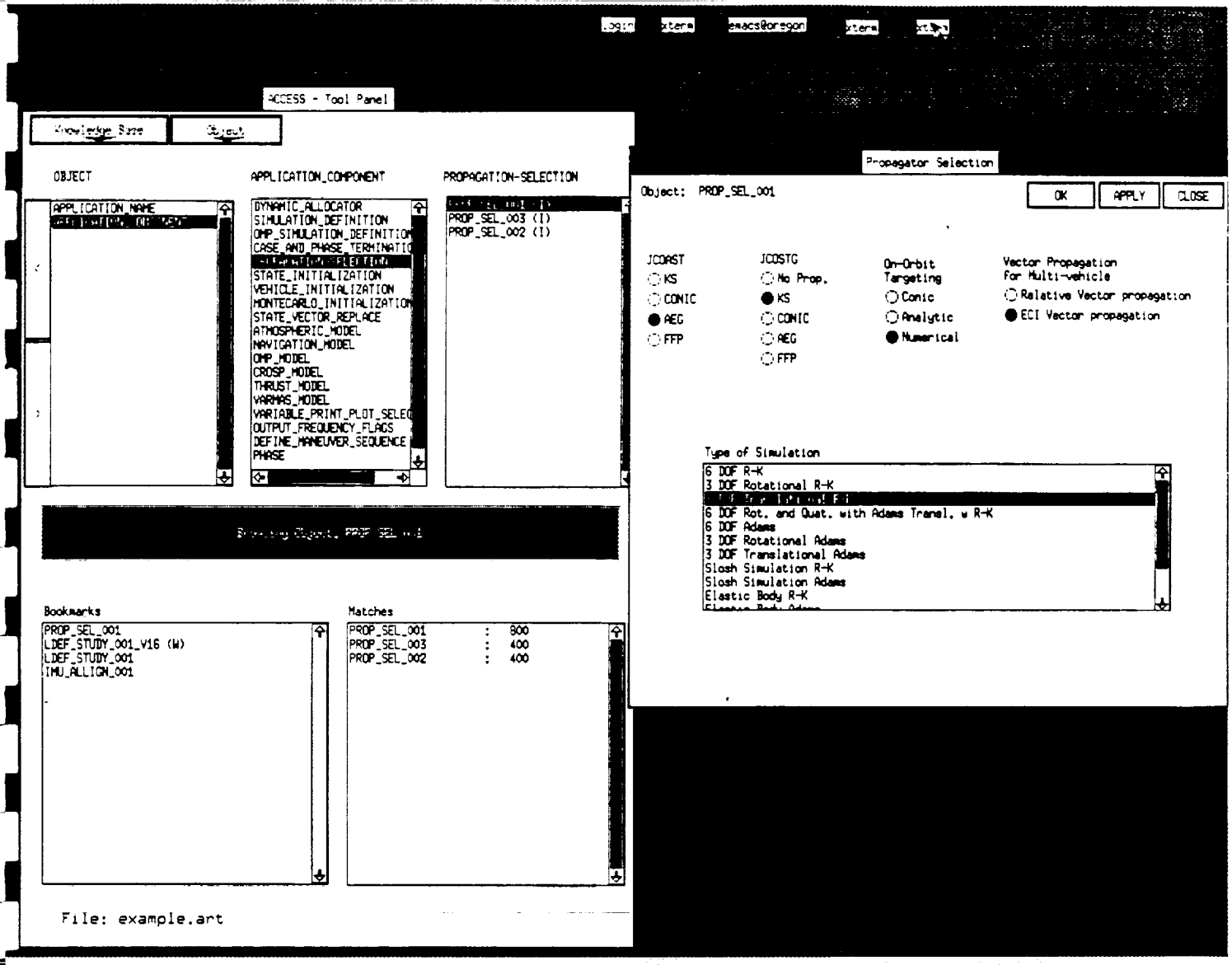
```
(reorder-values ?art-sequence ?ordering-function)
```

4.2 Specification of Forms in ACCESS

ACCESS uses TAE Plus (Transportable Applications Environment Plus) to support its user interface. TAE Plus provides a graphical, point and click user interface based on the X Window System. The knowledge engineer can use the TAE Plus Workbench to develop custom forms for browsing and editing objects in the knowledge base. The knowledge engineer must then specify the interface between these forms and the knowledge base. This section describes the structure of custom forms for use by ACCESS and describes that interface. For information on creating TAE forms, see the **TAE Plus User Interface Developer's Guide**.

4.2.1 Structure of Custom Forms

Figure 4-1: Example of a Custom Form

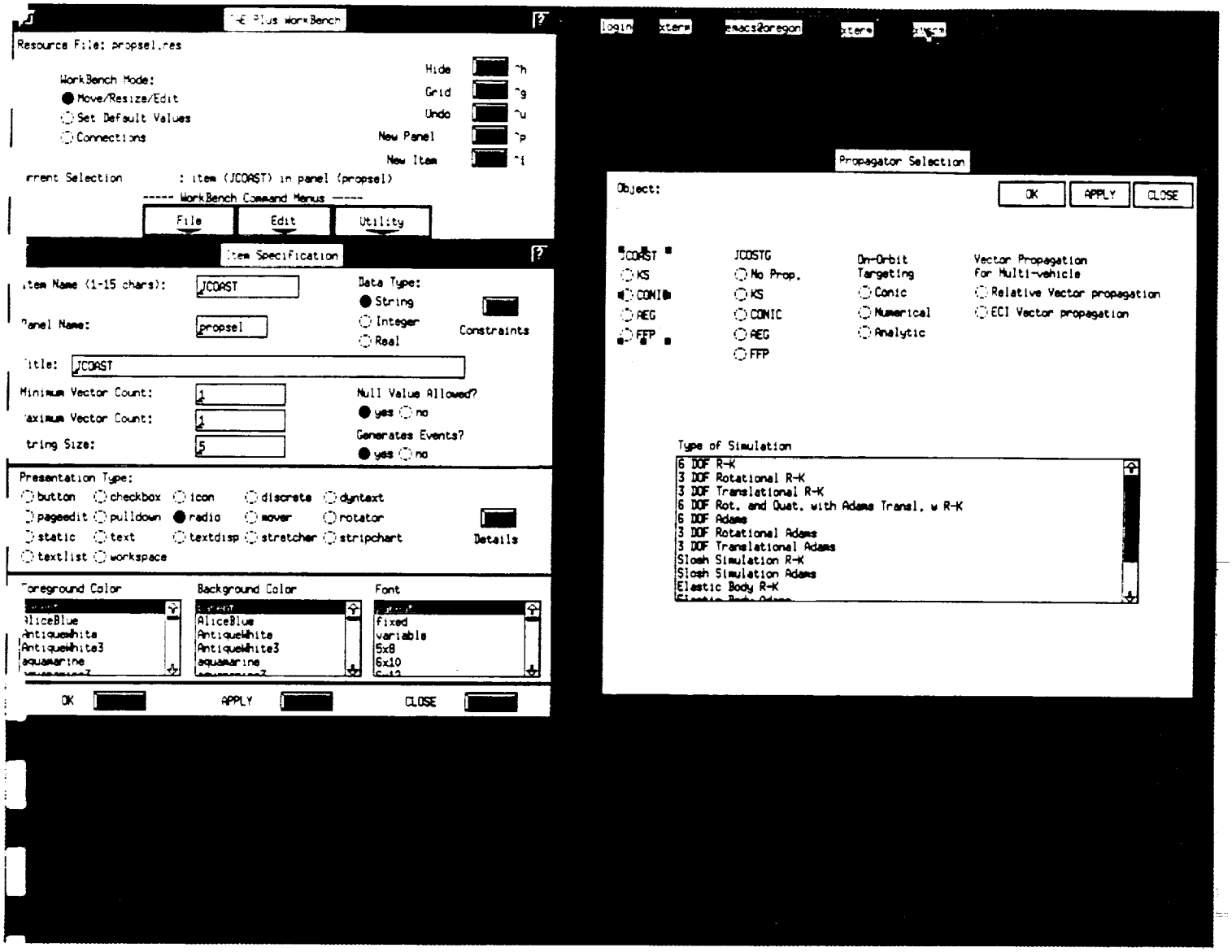


An example of a custom TAE form is shown in Figure 4-1.

A custom form is used to provide data entry for a single class of ACCESS objects. A form may consist of one or more TAE panels. If more than one panel is used, they should be positioned so that the end user can clearly see parts of all of them. These panels consist of "items", each of which has a data type and a presentation type.

Figure 4-2: Creating a Custom Form Using TAE Plus

ORIGINAL PAGE IS
OF POOR QUALITY



These panels may be created using the TAE Workbench utility. Figure 4-2 illustrates the use of the Workbench to edit the JCOAST item on the propsel panel. Note that this item is of data type string and presentation type Radio Button.

Every form which is intended to be used by ACCESS must have at least one panel which contain three items which have presentation type Button and whose names are "apply", "close", and "ok." When the user selects "apply", the information which has

been entered onto the form is stored in the corresponding object in the knowledge base. Selecting "close" causes the form to be removed from the screen and control returned to the main ACCESS Tools Panel. "ok" is a combination of "apply" and "close."

In addition to these three items, each forms panel must also contain a Static item whose name is "object." This item is used by ACCESS to display the name of the object which is being edited.

In addition to these required items, each panel may contain items which are used to enter attribute values. Each item on a form panel must correspond to an attribute in the class associated with the form. Items may be of type Radio, Checkbox, Text, Textlist, or Pagedit. The first four of these types may be used to supply a single value for an attribute. The last type (Pagedit) may be used to suply multiple values for an attribute (e.g., for an attribute corresponding to a multi-valued slot in ART-IM).

Items on the form correspond to TAE variables of type Real, Integer, or String. TAE variables of type Real are converted to ART-IM objects of type FLOAT; variables of type Integer are converted to ART-IM integers. TAE variables of type String may be converted to ART-IM objects of type string, float, integer, or symbol, depending on what sort of conversion is specified.

TAE supplies a built-in capability for providing help on individual panel items. For details on this capability, see the TAE documentation.

4.2.2 Interface Between TAE Custom Forms and ACCESS OBJECTS

The TAE specification for a custom form resides in a .res file. The mechanism for specifying the correspondence between the TAE form and ACCESS objects is ART-IM schema definitions. There are three principal types of schemas used: the **form-specification** schema, the **panel-spec** schema, and the **item-specification** schema. The definitions of these schemas are as follows:

```
(defschema has-item-specs
        (instance-of slot)
        (cardinality multiple))

(defschema conversion-specs
        (instance-of slot)
        (cardinality multiple)
        )

(defschema form-specification
        (file-name)     ;string
        (form-for-class);name of class to which this form applies
        (collection)    ;pointer supplied by ACCESS
        (has-panel-specs));sequence of panel-spec schemas
```

```
(defschema panel-spec
        (has-item-specs)
        (panel-name)     ;string
        (has-item-specs);names of item-specification schemas
        (target-pointer);pointer supplied by ACCESS
        (view-pointer)  ;pointer supplied by ACCESS
        (saved-view-pointer);pointer supplied by ACCESS
        (panel-id))      ;pointer supplied by ACCESS

(defschema item-specification
        (corresponds-to-attribute);name of attribute in class to which
                        [this item corresponds
        (parm-name)      ;string giving name of item in TAE
        (value-type)     ;value type in ART-IM - symbol, string,
                         ;(both corresponding to TAE strings)
                         ;integer (TAE integer), float (TAE real)
        (conversion-specs);OPTIONAL - two element sequences specifying
                         ;conversion from TAE strings to ART-IM symbols
        )
```

4.2.2.1 Form-specification Schemas

For each form created, the knowledge engineer must complete a **form-specification** schema as follows:

1. The value of the **file-name** slot of this schema must be a string containing the name of the .res file in which the form is specified.

2. The value of the **form-for-class** slot of this schema must be a symbol representing the class of ACCESS objects which correspond to this form.

3. The value of the **has-panel-specs** slot is a sequence, each element of which is the name of a **panel-spec** schema. There is one such schema for each panel comprising the form.

For each **panel-spec** schema referred to in the **form-specification** schema, the knowledge engineer must complete slots as follows:

1. The **panel-name** slot of this schema must be a string containing the TAE panel name.

2. The **has-item-specifications** slot is a multi-valued slot whose values are the name of **item-specification** schemas. There must be one such schema for each optional item on the panel.

The remaining slots of the **form-specification** and **panel-spec** schemas are used internally by ACCESS to store pointers to TAE structures.

The following is an example of completed **form-specification** and **panel-spec** schemas, corresponding to the form displayed in Figure 4-1.

```
(defschema form-for-propsel
        (instance-of form-specification)
        (file-name "forms.res")
        (form-for-class propagation-selection)
        (has-panel-specs (panel-for-propsel)))

(defschema panel-for-propsel
        (instance-of panel-spec)
        (panel-name "propsel")
        (has-item-specs jcoast-spec jcostg-spec on-orbit-targeting-spec
        vector-propagation-spec type-of-simulation-spec))
```

4.2.2.2 Item-specification Schemas

For each optional item on a custom form, the knowledge engineer must construct an **item-specification** schema which describes the interface between that item and the corresponding attribute in an ACCESS object.

Each **item-specification** schema must be created as follows:

1. The value of the **corresponds-to-attribute** slot is the name of the attribute in the ACCESS class which corresponds to this item. This name is actually the name of an ART-IM slot.

2. The value of the **parm-name** slot is a string corresponding to the name of the TAE item.

3. The value of the **value-type** slot specifies the manner in which the value on the form will be converted from a TAE variable to an ART-IM object. Allowable values for this slot are as follows:

    a. `FLOAT`. This is used when the item on the TAE panel is of data type Real. The value entered on the form is then converted from a TAE Real to an ART-IM float and vice-versa.

    b. `INTEGER`. This is used when the item on the TAE panel is of data type Integer. The value entered on the form is then converted from a TAE integer to an ART-IM integer and vice-versa.

    c. `STRING`. This may be used when the item on the TAE panel is of data type String. The value entered on the form is then converted from a TAE String to an ART-IM string and vice-versa. This is normally used for TAE items with presentation type Text, which allow free-form text entry.

    d. `SYMBOL`. This may be used when the item on the TAE panel is of data type String. If there are no values in the **has-conversion-specs**

slot of the schema, then the value entered on the from is converted from a TAE string to an ART-IM symbol using the function a_read_from_string. Conversion from an ART-IM string to a TAE string is done using the function a_symbol_value. a_read_from_string will return a symbol whose name is represented by a string of upper case characters and which is terminated by the first white space in the TAE string. Thus to ensure that the resulting ART-IM symbol can be correctly converted back to the original TAE String, this form of conversion should be used only for RADIO button items or TEXTLIST items whose valid strings are entirely upper case.

If there are values in the **has-conversion-specs** slot, then the program will first try to use these specifications to convert from a TAE string to an ART-IM object and vice-versa. The algorithm for conversion is described below.

e. MIXED. This may be used when the item on the TAE panel is of type String. This is normally used when the TAE item has presentation type Radio Button item or Textlist. In this case, ACCESS uses the values in the **has-conversion-specs** slot to try to make a conversion from a TAE string to an ART-IM object. Each value in the **has-conversion-specs** slot is a two element sequence, whose first element is a string and whose second element is an arbitrary ART-IM object. A TAE string is converted to an ART-IM object by searching for a sequence whose first element matches the TAE string and then taking the second element of the sequence. If no string is found matching the TAE string, an error message is returned and the corresponding form is erased from the screen. Thus the knowledge engineer must be very careful to provide correct conversion specifications.

The following example shows an **item-specification** schema which is used to specify the conversion from a Radio button item to an attribute with integer values:

```
(defschema jcoast-spec
        (instance-of item-specification)
        (corresponds-to-attribute jcoast)
        (parm-name "JCOAST")
        (value-type MIXED)
        (conversion-specs ("KS" 1) ("CONIC" 2) ("AEG" 3) ("FFP" 4))
        )
```